Artificial Neural Networks and Deep Learning

LaTeX by Min Soo Jeon
mjeon@uc.cl
Professors Matteo Matteucci and Giacomo Boracchi
Politecnico di Milano

Semester 2024-2

Disclaimer:

This material is an unofficial resource created for educational purposes only. The authors and contributors do not assume any liability for errors, inaccuracies, or misinterpretations that may arise from its use. Readers should verify information from official sources.

Version 1.0.0

Introduction

These notes are based on the lectures delivered by Professors Matteo Matteucci and Giacomo Boracchi during the Artificial Neural Networks and Deep Learning course at Politecnico di Milano during Semester 2024-2. Official course information can be found on the official websites of Professors Matteuci (Matteucci, 2024) and Boracchi (Boracchi, 2024). Some sections, particularly the practical components, were taught by Teaching Assistant Eugenio Lomurno. Further details on the practical part can be found in the Lomurno's notebooks (Lomurno, 2024).
This material may be outdated, the latest version is available in the repository. To report an error, please contact mjeon@uc.cl.

Contents

1 Machine Learning vs Deep Learning

1.1 Machine Learning

It is a field of research and type of algorithms that focus on finding patterns in data to make predictions. It completes its task using hand-crafted features, in simpler terms, features that are not extracted directly, but are built and/or chosen by a third party
We define:

Definition 1.1: T

The task that we will run

It could be regression, classification, and others

Definition 1.2: D

Data := $x_1, x_2, x_3, \ldots, x_N$

Definition 1.3: E

Error or Loss

Supervised Learning Given desired outputs, learn how to produce the correct output given a new set of inputs

Classification Learning how to determine which category an instance is part of

Regression Learning how to determine which value an instance has, based on a feature.

Unsupervised Learning Exploit regularities in D to build a representation to be used for a certain task

Clustering Learning how to split the data into groups or clusters, without necessarily knowing, what each one means.

Reinforcement Learning Producing actions $a_1, a_2, a_3, \ldots, a_N$ which affect the environment, and receiving rewards $r_1, r_2, r_3, \ldots, r_N$, learning to act to maximize rewards in the long term. *This course will not be focused on this last one*

1.2 Deep Learning

It is a field of research and type of algorithms that focus on learning from raw data. Deep Learning is about data representation, that is, about how to represent the data in order to perform the task with a better input.

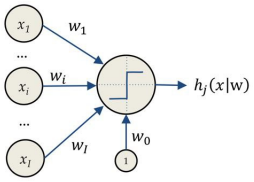
1.3 Which is better

Both, machine learning and deep learning are important to perform tasks.
Machine Learning algorithms usually require fewer computational resources and need well-built pre-existing features, i.e., an expert who knows how to build them. It is often used for smaller datasets, due to the difficulty of doing this task for huge amounts of data.
Deep learning algorithms usually take more computational resources. The big advantage is that it could create its own data representation, solving a double task. Plus, the model could find high-level features (those that are not easily identified from raw data).

2 Basic Feed Forward Neural Networks (FFNN)

2.1 Perceptron: The introduction

The perceptron is a computational model based on a brain, taking the concept of neurons and their synaptic connections to perform a huge number of computing units.
Let b be a "bias", and $x_1, \ldots, x_I$ signals from previous steps. In the context of the perceptron, a neuron is a unit that receives $1, x_1, \ldots, x_I$, each one with weights $b, w_1, \ldots, w_I$. The output j of this neuron is a function $h_j = sign()$ applied to the weighted sum of x_i's. (see figure 1)



$$h_j(x|\mathrm{w},\mathrm{b}) = \mathrm{h_j}\left(\Sigma_{i=1}^{I} w_i \cdot x_i - b\right) = \mathrm{h_j}\left(\Sigma_{i=0}^{I} w_i \cdot x_i\right) = h_j(w^T x)$$

Figure 1: Basic neuron diagram

We notate $b = w_0$ by convention

Perceptron can be used to model logical NOT, AND, and OR. You can check that OR with 2 literals can be modeled with $w_0 = -1/2, w_1 = 1, w_2 = 1$, while AND can be modeled with $w_0 = -2, w_1 = 3/2, w_2 = 1$.

This is relevant because any boolean function can be approximated using these three logical operations, due to the completeness of DNF formulas. See this article (Wikipedia contributors, 2024a).

Given this, we know that there exist weights that approximate the boolean function we want to learn, but we still don't know how to find those weights. For learning the weights that best approximate the function, we use Hebbian Learning rule:

$$w_i^{k+1} = w_i^k + \Delta w_i^k$$
$$\Delta w_i^k = \eta \cdot x_i^k \cdot t^k$$

where:
η : learning rate

x_i^k : the i^{th} perceptron input at time k
t^k : the desired output at time k

Given the characteristics of the perceptron, it can be interpreted as a linear classifier, for which the decision boundary is the hyperplane

$$w_0 + w_1 \cdot x_1 + \cdots + w_I \cdot x_I = 0$$

In particular, in the 2D case,

$$x_2 = -\frac{w_0}{w_2} - \frac{w_1}{w_2} \cdot x_1$$

One of the limitations of the perceptron is that it can only model linear separable classification data.

2.2 Idea of FFNN's

Feed-forward neural Networks are a generalization of the perceptron. They consist of multiple perceptrons interconnected by weights, with more general (although differentiable) functions h_j. It could be seen as in figure 2

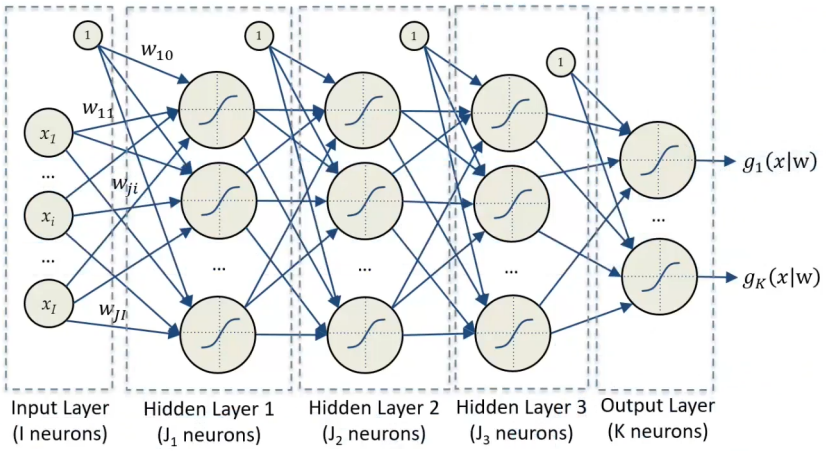


Figure 2: Simple Feed Forward Neural Network Diagram

Neurons could be sorted into different layers, related to which weights they receive and propagate. This gives order and allows to computate back propagation algorithm, the Hebbian Learning Rule of FFNN.
The first neurons are the ones that receive the data, its set is called Input Layer. Then, there are some layers that are not the final output, but they give more complexity to the model. Finally, there is the Output Layer, that will provide the model's response to the task. In its most elemental way, every layer i is connected to the layer i-1 and i+1. This is a general structure of an FFNN, but there also exist shortcut connections, that connect a neuron with another skipping some layers in the middle.
If a Neural Network is fully connected, in other words, each layer i and $i+1$ have all its

connections, the number of weights between these 2 layers, is $|J_i| \times |J_{i+1}|$, where $|J_i|$ is the number of neurons in layer i

2.3 Activation Function

We call Activation Functions to the functions that are applied over the weighted sum of inputs in a neuron, h_j in Perceptron.

The importance of activation functions lies in the non-linearity we can add to the algorithm. Plus, these functions have to be differentiable to apply the gradient descent.

There are a lot of activation functions, we introduce some of the most used:

- Identity: Since spans the whole $\mathbb{R}$, used in regression output layer

$$g(a) = a$$
$$g'(a) = 1$$

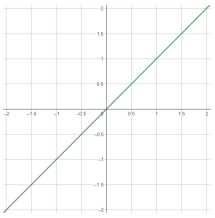


Figure 3: Identity activation function

- Sigmoid: Since goes from 0 to 1, used in 0-1 classification classes and probabilities of belonging to a class as well. It is also useful for K-multiclass classification since K neurons with sigmoid allow you to assign a high probability for K neurons independently.

$$g(a) = \frac{1}{1 + e^{-a}}$$
$$g'(a) = g(a)(1 - g(a))$$

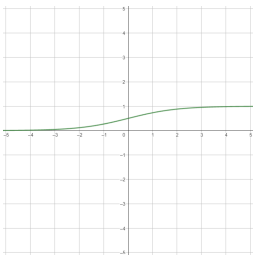


Figure 4: Sigmoid activation function

- Softmax: Since K neurons map to $[0, 1]^K$, satisfying $\sum_k^K g(a_k) = 1$, it is used for multi-classes output layer when classes are coded with one hot encoding (Wikipedia

contributors, 2024b).

$$g(a_k) = \frac{e^{a_k}}{\sum_k e^{a_k}}$$
$$\frac{\partial g(a_k)}{\partial a_j} = a_k \cdot (\mathbf{1}_{\{i=j\}} - a_j)$$

- Tanh: Since goes from -1 to 1, used in -1 - 1 classification classes

$$g(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}$$
$$g'(a) = 1 - g(a)^2$$

Figure 5: hyperbolic tangent activation function

When we want to use sigmoid or tanh as activation functions for hidden layers, it appears a problem. The gradient in those functions is close to 0 in some sections, and always less than 1. As we will see in section 2.5.3, backpropagation requires gradient multiplications. When we have deep neural networks (with lots of layers), the gradients far away from the output vanish. This is called the vanishing gradient phenomenon. This does not permit the Neural Network to learn quickly (or even learn).
Remark: This phenomenon is well-known also in Recurrent Neural Networks, architecture introduced in section 23

Very big gradients are also a problem, because the gradient will grow too much for faraway layers. This is why we want our gradients as close to 1 as possible.
There are some well-known gradients used for hidden layers:

- Relu: Very often in hidden layers

$$g(a) = max(0, a) = \begin{cases} a & \text{si } a > 0 \\ 0 & \text{otherwise} \end{cases}$$
$$g'(a) = \mathbf{1}_{a>0} = \begin{cases} 1 & \text{si } a > 0 \\ 0 & \text{otherwise} \end{cases}$$

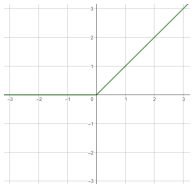


Figure 6: Relu activation function

It has advantages:

1. Up to 6x faster SGD convergence than sigmoid/tan.
2. Sparse Activation, only part of the units are activated
3. Efficient gradient propagation (no vanishing or exploding gradient problems)
4. Scale invariant: $max(0, ax) = a \cdot max(0, x)$

But also some disadvantages:

1. Non-differentiable at zero
2. Non-zero centered output
3. Unbounded: could potentially blow up
4. Dying neurons: No gradients flow backward through the neuron, so it becomes stuck and "dies". It decreases model complexity, especially with high learning rates

- Leaky Relu: Fix for the dying neurons problem

$$g(a) = \begin{cases} a & \text{if } a \geq 0 \\ 0.01a & \text{othewise} \end{cases}$$

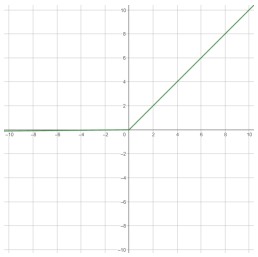


Figure 7: Leaky Relu activation function

- ELU: Try to make the mean activations closer to zero, which speeds up learning, adding a hyperparameter

$$g(a) = \begin{cases} a & \text{if } a \geq 0 \\ \alpha(e^a - 1) & \text{othewise} \end{cases}$$

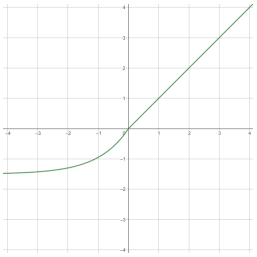


Figure 8: ELU activation function

- There are other activation functions like SiLU, Swish, E-Swish, FTS, T_rec, Softplus, among others

2.4 Universal Approximation of Neural Networks

The universal approximation theorem of ANN states: "A single hidden layer feedforward neural network with S shaped activation functions can approximate any measurable function to any desired degree of accuracy on a compact set". Mathematically, the family of neural networks is dense in the function space.
It is important to consider that:

1. In the worst case, an exponential number of hidden units may be required
2. It doesn't mean that an algorithm can find the correct weights
3. The layer may have to be unfeasibly large and may fail to learn and generalize

Classification only requires one extra layer

2.5 Optimization

We already know how the model works and what calculation it does to give an output. We also know that there exist weights that can lead this process to a determined task, but we still don't know how to know what the weights should be. Back Propagation is an optimization process to approximate these weights.

2.5.1 Error function

Basically, given a training set D, we want to find model parameters such that for new data z_n

$$g(z_n|w) \sim t_n$$

Then (We will dive into this later), we want to minimize the loss or error function:

$$E(z_n, w) = E(w) = \sum_{n=1}^{N} (t_n - g(z_n|w))^2$$

2.5.2 Gradient Descent

If we want to minimize this error through a linear function, we can find the model with the sum of squared errors. Anyway, our model is not linear.

Another thought could use the first order criteria, in other words, derive and equal to 0. The problem is that with so many parameters, solving the system of equations requires too much computational power.

Finally, we note that we can use iterative methods. In particular, we will use gradient descent, which consists of updating the model parameters with the following rule.

$$w^{k+1} = w^k - \eta \frac{\partial E(w)}{\partial w}\bigg|_{w^k}$$

where η is the learning rate.
A common problem of the classic gradient descent is getting stuck in the local minima. There is a modified version of it, Gradient Descent with Momentum, that can help to make this problem less often.

$$w^{k+1} = w^k - \eta \frac{\partial E(w)}{\partial w}\bigg|_{w^k} - \alpha \left.\frac{\partial E(w)}{\partial w}\right|_{w^{k-1}}$$

where α is the momentum constant.

Taking this idea, there was developed another variant, the Nesterov Accelerated gradient, that proposed making a jump as momentum and then adjusting the value

$$w^{k+\frac{1}{2}} = w^k - \alpha \left.\frac{\partial E(w)}{\partial w}\right|_{w^{k-1}}$$
$$w^{k+1} = w^k - \left.\frac{\partial E(w)}{\partial w}\right|_{w^{k+\frac{1}{2}}}$$

There are a lot of variations of gradient descent as well, each one taking advantage of certain characteristics of the objective function and also dealing with certain problems that could appear. Rprop, AdaGrad, SGD+Rprop, AdaDelta, Adam, and AdamW are between them.

For example, some of them take the idea that in each layer, neurons can learn differently. Therefore, we can use separate adaptive learning rates. This could solve, for example, the problem of vanishing gradients.

RMSProp (Root Mean Square Propagation), combined with Gradient Descent with Momentum, makes the optimizer that is most used today for Machine Learning, Adam.
Learning Rate Really matters, the time to achieve the function varies a lot.

Due to computational power issues, we might not want to execute gradient descent with all the examples, as it does batch gradient descent:

$$\frac{\partial E(w)}{\partial w} = \frac{1}{N}\sum_{n}^{N} \frac{\partial E(x_n, w)}{\partial w}$$

An alternative could be to do just one sample, unbiased with high variance, this is called

Stochastic Gradient Descent (SGD):

$$\frac{\partial E(w)}{\partial w} \approx \frac{\partial E_{SGD}(w)}{\partial w} = \frac{\partial E(x_n, w)}{\partial w}$$

Another alternative could use all the data, but not at the same time. This is called Mini-Batch Gradient Descent:

$$\frac{\partial E(w)}{\partial w} \approx \frac{\partial E_{MB}(w)}{\partial w} = \frac{1}{M} \sum_{n \in \text{Minibatch}}^{M<N} \frac{\partial E(x_n, w)}{\partial w}$$

2.5.3 Chain Rule

We can use the chain rule since Forward Propagation is basically a composition of functions. If we want to compute the derivative of each weight without repeating calculus:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx} = f'(y)g'(x) = f'(g(x))g'(x)$$

Plus, when we pass through the forward process, we can evaluate the local derivatives (closed expressions if we choose correct activation functions) and store them locally in each unit. In this way, we can use these values in the backward pass.
This whole process is 8x more efficient in gpu than cpu

3 Loss or Error Function

3.1 Common Losses

Some of the most common losses are the following:

- **MSE**: Mean Square Error is used for continuous prediction tasks

$$\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

- **Binary Cross-Entropy**: Used for binary class classification

$$-\sum_{i=1}^{N}(y_i log(\hat{y}_i) + (1 - y_i) log(1 - \hat{y}_i))$$

- **Categorical Cross-Entropy**: It receives one-hot vectors. Used for multiclass classification

$$-\sum_{i=1}^{N} y_i \log(\hat{y}_i)$$

- **Sparse Categorical Cross-Entropy**: Similar to Categorical Cross-Entropy, but the labels are integers instead of one-hot vectors

3.2 Maximum Likelihood Estimation

Given data with a certain distribution, we want to find the parameters (or the hypothesis) that make most of the points likely to be observed.
In other words, we want: Letting $\theta = (\theta_1, \theta_2, \ldots, \theta_p)^T$ be a vector of parameters. Finding the MLE for θ. Then we have to follow these steps:

1. Write the likelihood $L = P(Data|\theta)$
2. If it simplifies the expression, take the logarithm $l = \log P(Data|\theta)$
3. Solve the first-order criteria equations
4. Check that θ^{MLE} is a maximum

3.2.1 MSE

In the case of regression, we will infer the MSE from the next case. Let's observe i.i.d. samples from Gaussian Distribution with known σ^2. In other words:

$$x_1, x_2, \ldots, x_N \sim N(\mu, \sigma^2)$$

$$p(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

So we can calculate the MLE

$$\begin{aligned}
L(\mu) &= \prod_{n=1}^{N} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x_n-\mu)^2}{2\sigma^2}} \\
l(\mu) &= \log\left(\prod_{n=1}^{N} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x_n-\mu)^2}{2\sigma^2}}\right) \\
&= N \log \frac{1}{\sqrt{2\pi}\sigma} - \frac{1}{2\sigma^2} \sum_{n=1}^{N} (x_n - \mu)^2 \\
\frac{\partial l(\mu)}{\partial \mu} &= \frac{\partial}{\partial \mu}\left(N \log \frac{1}{\sqrt{2\pi}\sigma} - \frac{1}{2\sigma^2} \sum_{n=1}^{N} (x_n - \mu)^2\right) \\
&= \frac{1}{2\sigma^2} \sum_{n=1}^{N} 2(x_n - \mu) \\
\implies & \frac{\partial l(\mu)}{\partial \mu}(\mu^{MLE}) = 0 \\
\implies & \frac{1}{2\sigma^2} \sum_{n=1}^{N} 2(x_n - \mu^{MLE}) = 0 \\
\implies & \mu^{MLE} = \frac{1}{N} \sum_{n=1}^{N} x_n
\end{aligned}$$

Now, in the case of Neural Network, we want to approximate a target function t having N observations:

$$t_n = g(x_n|w) + \epsilon_n, \qquad \epsilon_n \sim N(0, \sigma^2)$$

We want to obtain the MLE of t. Since $t \sim N(g(x|w), \sigma^2)$ and following the previous process:

$$l(w) = N \log \frac{1}{\sqrt{2\pi}\sigma} - \frac{1}{2\sigma^2} \sum_{n=1}^{N} (t_n - g(x_n|w))^2$$

Then, we want to maximize the loglikelihood, this is equivalent to :

$$argmax_w l(w) = argmin_w \sum_{n=1}^{N} (t_n - g(x_n|w))^2$$

Summarizing, this is the expression that we want to find if we want to fit a normal distribution to our data.

3.2.2 Binary Cross-Entropy

In the case of classification, we will infer the Binary Cross-Entropy from the next case. Now, the goal is to approximate a posterior probability t having N observations.

$$g(x_n|w) = p(t_n|x_n), t_n \in \{0, 1\} \implies t_n \sim Be(g(x_n|w))$$

Then, having i.i.d. samples:

$$p(t|g(x|w)) = g(x|w)^t \cdot (1 - g(x|w))^{1-t}$$

Repeating the process mentioned before:

$$l(w) = \sum_{n=1}^{N} t_n \log g(x_n|w) + (1 - t_n) \log(1 - g(x_n|w))$$

Then, the weights for maximizing the loglikelihood are:

$$argmin_w - \sum_{n=1}^{N} t_n \log g(x_n|w) + (1 - t_n) \log(1 - g(x_n|w))$$

The expression could remind to Categorical Cross-entropy:

$$- \sum_{n=1}^{N} t_n \log g(x_n|w)$$

3.3 Choosing the error function

For designing error functions:

1. Use all your knowledge/assumptions about the distribution data
2. Exploit background knowledge on the task and the model

3.4 Hyperplanes Linear Algebra

Let's consider the hyperplane (affine set) $\mathbf{L} \in \mathbb{R}^2$

$$L : w_0 + w^T x = 0$$

For any two points x_1 and x_2 on $\mathbf{L} \in \mathbb{R}^2$ we have:

$$w^T(x_1 - x_2) = 0$$

The versor (unitary vector) normal to $\mathbf{L} \in \mathbb{R}^2$ is then

$$w^* = w/\|w\|$$

And for any point x_0 in $\mathbf{L} \in \mathbb{R}^2$ we have

$$w^T x_0 = -w_0$$

The signed distance of any point x from $\mathbf{L} \in \mathbb{R}^2$ is defined by

$$d := w^{*T}(x - x_0) = \frac{1}{\|w\|}(w^T x + w_0)$$

Finally, we can note that $(w^T x + w_0)$ is proportional to the distance of x from the plane defined by $w^T x + w_0 = 0$

Let's see that the Hebbian Rule is the result of minimizing the error function that uses the distance of misclassified points from the decision boundary
Remember that a misclassified point in Perceptron satisfies $t_i(w^T x_i + w_0) < 0$. Then, the goal is minimizing:

$$D(w, w_0) = -\sum_{i \in M} t_i(w^T x_i + w_0)$$

This is non-negative and proportional to the distance of the misclassified points from $w^T x + w_0 = 0$
Let's minimize by stochastic gradient descend the error function:

$$\frac{\partial D(w, w_0)}{\partial w} = -\sum_{i \in M} t_i x_i \qquad \frac{\partial D(w, w_0)}{\partial w_0} = -\sum_{i \in M} t_i$$

Then the stochastic gradient descent applies for each misclassified point:

$$\begin{pmatrix} w^{k+1} \\ w_0^{k+1} \end{pmatrix} = \begin{pmatrix} w^k \\ w_0^k \end{pmatrix} + \eta \begin{pmatrix} t_i x_i \\ t_i \end{pmatrix} = \begin{pmatrix} w^k \\ w_0^k \end{pmatrix} + \eta \begin{pmatrix} t_i x_i \\ t_i x_0 \end{pmatrix} \quad \text{(Remember } x_0 = 1\text{)}$$

Finally, we can see that Hebbian Learning is just a particular case of Stochastic Gradient Descent

4 Neural Networks Training and Overfitting

Overfitting consists of a characteristic of a model that adjusts excessively to the data. This means that the model performs well for the training data, but it is not able to transfer this "knowledge" to new data, making it useless.

4.1 Model Complexity

Inductive Hypothesis: A solution approximating the target function over a sufficiently large set of training examples will also approximate it over unobserved examples.

Model Complexity (Intuition): Number of possible datasets that the model can approximate

When a model doesn't have enough complexity, it can't predict well. It is said that it underfits the data.

In contrast, when a model have too much complexity, it doesn't generalize well. It is said that it overfits the data.

4.1.1 How to measure generalization

From what we have seen, training error or loss, is not a good indicator of performance on future data, since the classifier has been trained from the training data, and the new data will not be the same as the training data.
We need to test generalization on an independent new test set. This could be done in a later evaluation of the model. Otherwise, if we cannot lose data, since we are working with lacking data, we can perform random subsampling with replacement on the dataset. There are some techniques used for this, such as bootstrap or jackknife.
In classification, it is recommended to preserve class distribution, i.e., with stratified sampling

4.2 Training process

There are some terms used in training context, just as in figure 9

Definition 4.1: Training Dataset

The available data

Definition 4.2: Training set

The data used to learn model parameters

Definition 4.3: Test set

The data used to perform final model assessment

Definition 4.4: Validation set

The data used to perform model selection

Definition 4.5: Training data

The data used to train the model (fitting + selection)

Definition 4.6: Validation data

The data used to assess the model quality (selection + assessment)

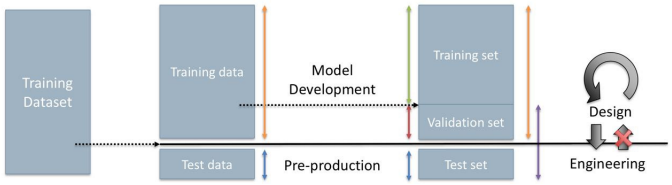


Figure 9: Types of data used in Machine Learning

Definition 4.7: Epoch

Consists of one stage of the model. It is completed when all the training data is passed through the training process

This is regardless of the type of gradient descent used (see section 2.5.2). We need one iteration per epoch only when we use batch gradient descent

Cross Validation is the use of the training dataset to both train the model (parameter fitting + model selection) and estimate its error on new data.

- When lots of data are available, use a Hold Out set and perform validation
- When having little data available, use Leave-One-out Cross-Validation (LOOCV). It consists of leaving the $n \in \{1, \ldots, N\}$ data instances out of the model, training without it, and validating it in that single case.
- K-fold Cross Validation is a good trade-off (sometimes better than LOOCV). It consists of doing k partitions where the model has to be trained and validated every time.

In the cases where we have more than one model, we must choose a final one. We can:

- Choosing the one with the best validation set
- Choosing the average
- Using all the models and deciding by voting

When we have more than one model, we compute our model loss as:

$$\hat{E} = \frac{1}{K}\sum_{k}\hat{e}_k \qquad \hat{e}_k = \frac{1}{|N_k|}\sum_{n_k \in N_k} E(x_{n_k}|w)$$

4.3 Limiting Overfitting by Cross-validation

Overfitting networks show a monotone training error trend on average with SGD as the number of gradient descent iterations k grows. The problem is that the model loses generalization at some point.
To solve this, we can use Early Stopping technique (see figure 10):

1. Hold out some data
2. Train on the training set
3. Perform cross-validation on the hold-out set
4. Stop train when validation error does not improve in *patience* epochs, being *patience* a variable

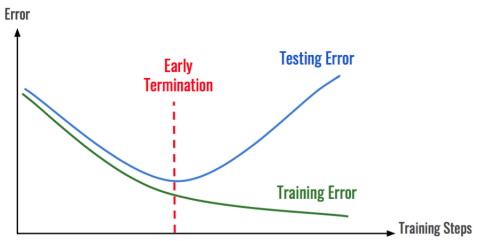


Figure 10: Error vs iterations. Training and test error

This helps with both less time for training and avoiding overfitting. When we have 'enough' data, early-stopping technique is enough to prevent overfitting
For the final training, when we use both training and validation set for training the final model, it is recommended to set the maximum of epochs as Early Stopping marked before

Model selection and evaluation happens at different levels. The one we already talked about, Parameter Level, i.e., when we learn the weights w for a neural network. But we are also selecting at a Hyperparameter Level, i.e., when we choose the number of layers L or the number of hidden neurons $J^{(l)}$ for a given layer. We can use Early Stopping for choosing hyperparameters as well (see figure 11), choosing the hyperparameter with the best validation error.

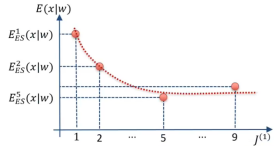


Figure 11: Error vs iterations. Training and test error

Remark: Every knowledge we acquired with Cross-validation, should be fixed in the final model.

4.4 Weight Decay or Ridge Regression

We can limit overfitting by making small weights more likely. Thus, Weight regularization is about constraining the model freedom, based on a priori assumption on the model, to reduce overfitting.
Previously, we have maximized the data likelihood:

$$w_{MLE} = argmax_w P(D|w)$$

Now, we can assume a priori distribution on parameters distribution, and we will calculate the maximum probability a posteriori.

$$\begin{aligned} w_{MAP} &= argmax_w P(w|D) \\ &= argmax_w P(D|w) \cdot P(w), \qquad P(w) \sim N(0, \sigma_w^2) \end{aligned}$$

Empirical, small weights have been observed to improve the generalization of ANN.
Now, if we want to maximize the likelihood, we follow the same process as before:

$$\hat{w} = argmax_w P(D|w)P(w) = argmin_w \sum_{n=1}^{N} \frac{(t_n - g(x_n|w))^2}{2\sigma^2} + \sum_{q=1}^{Q} \frac{w_q^2}{2\sigma_w^2}$$

$$= \underbrace{argmin_w \sum_{n=1}^{N} (t_n - g(x_n|w))^2}_{Fitting} + \underbrace{\gamma \sum_{q=1}^{Q} (w_q)^2}_{Regularization}$$

Then, we have another loss function, with a regularization. γ is the regularization parameter. When $\gamma \to 0$ the model is more likely to overfit the data. When $\gamma \to \infty$ the model is more likely to underfit the data, because the fitting summand loses importance in the optimization problem.
Again, you can use cross-validation to select the proper γ.
Note that minimization must be over the new loss function, but the model's evaluation should be done over the first one

Remark: Professor recommends using Early Stopping for big amounts of data, and regularization when there are fewer data.

4.5 Dropout

By turning off some neurons randomly, we force the rest to learn an independent feature, preventing co-adaptation (hidden units relying on other units) (see figure 12).

With this method, each hidden unit is set to zero with $p_j^{(l)}$ probability on each epoch, so then we train the model without considering the zero ones.

Dropout trains weaker classifiers, on different mini-batches. To rescale this effect, during the forward pass in the training, we re-scale the activation functions by $\frac{1}{1-p_j^{(l)}}$, so we increase

the expected sum of the neurons as we were in the case without dropout.

Finally, at inference time, both the weights and activation functions are used as if there were no dropout.

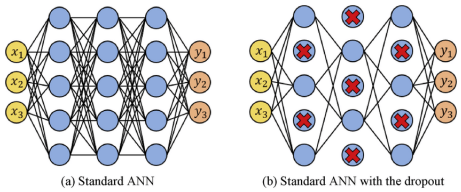


Figure 12: Example of neurons being dropped out

This technique could be applied to every layer of the network, or just to selected ones. The dropout generally varies up to 20% in input layers and up to 50% in hidden ones, depending on the aggressiveness of the dropout

4.6 Weights Initialization

The final result of gradient descent is affected by weight initialization, due to local minima and possible small gradients.
Some possible options are:

- Zeros: It does not work, all gradients would be zero and no learning would occur
- Big Numbers: It does not work, it could take too much to converge
- $w \sim N(0, 0.01)$: It is good for small networks, but it might be a problem for deeper neural networks

Some problems that we should take care of:

- If weights start so small, the gradient shrinks as it passes through each layer
- If weights start so big, the gradient grows as it passes through each layer until it is too massive to be used and manageable for the programming language

In order to solve these problems, there are some classic initializations proposed, Xavier and He initialization.

4.6.1 Xavier Initialization

Suppose we have an input x with I components and a linear neuron (Strong assumption, but linear function was very used in the past) with random weights w. Then, its output is

$$h_j = w_{j1}x_1 + \cdots + w_{ji}x_i + \cdots + w_{jI}x_I$$

It can be prove that

$$Var(w_{ji}x_i) = E[x_i]^2 Var(w_{ji}) + E[w_{ji}]^2 Var(x_i) + Var(w_{ji})Var(x_i)$$

Now, if inputs and weights both have mean=0 (It is recommended standardizes the inputs), that simplifies the expression to

$$Var(w_{ji}x_i) = Var(w_{ji})Var(x_i)$$

Plus, if we assume that all w_i and x_i are i.i.d.

$$Var(h_j) = Var(w_{j1}x_1 + \cdots + w_{ji}x_i + \cdots + w_{jI}x_I) = I \cdot Var(w_i)Var(x_i)$$

We can note that the variance of input is the variance of the input scaled by $I \cdot Var(w_i)$
Then, if we want the variance of the input and output to be the same, we have to set $I \cdot Var(w_j) = 1$
For this reason, Xavier proposes to initialize:

$$w \sim N\left(0, \frac{1}{n_{in}}\right)$$

Performing similar reasoning for the gradient (if you want that it has Variance=1), Glorot & Bengio found $n_{out}Var(w_j) = 1$. Then, to adapt his solution, Xavier proposed:

$$w \sim N\left(0, \frac{2}{n_{in} + n_{out}}\right)$$

4.6.2 He Initialization

More recently, He Proposed, for rectified linear units (Relu):

$$w \sim N\left(0, \frac{2}{n_{in}}\right)$$

Intuitively, with Relu half of the initial neurons would be 0, since they follow a normal distribution with mean 0, and everything under 0 is mapped to 0. He took this as an assumption.

4.7 Normalization

Usually, normalization is useful in gradient-based optimization, making the model more robust to bad initialization. This evolves normalization parameters (statistics) of the model.
Any preprocessing statistic calculated over more than 1 instance data must be computed on training data, but applied to all training, test, and validation data. It is important to ensure the model is behaving equally in training and predicting data.
There are different forms of normalizing data:

- Zero-center (most used): Subtract the mean and divide by the standard deviation. It could be channel-wise or along all the image
- Min-max: Subtract the min and divide by the max minus the min
- PCA: Decorrelate and whiten data.

Networks converge faster if input has been whitened (zero mean and unit variance) and are uncorrelated to account for covariate shift

We can have internal covariate shift, normalization could be useful also at the level of hidden layers.

Batch normalization is a technique to cope with this. We can create BatchNorm layers after fully connected layers (or convolutional layers that we will see in section 6), and before nonlinearities.

This could be interpreted as doing preprocessing at every layer of the network, but integrated in a differentiable way

Algorithm 1 Batch Normalization

Require: Values of x over a mini-batch: $\mathbf{B} = x_1, \dots, x_m$; Parameters to be learned: γ, β
Ensure: $\{y_i = BN_{\gamma,\beta}(x_i)\}$

$\mu_{\mathbf{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$ ▷ mini-batch mean
$\sigma^2_{\mathbf{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathbf{B}})^2$ ▷ mini-batch variance
$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathbf{B}}}{\sqrt{\sigma^2_{\mathbf{B}} + \epsilon}}$ ▷ normalize
$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv BN_{\gamma,\beta}(x_i)$ ▷ scale and shift

It has been shown that batch normalization can improve the gradient flow through the network. Plus, it allows using higher learning rates and reduces the strong dependence on weights initialization. In addition, it acts as a form of regularization, slightly reducing the need for dropout.

There also exist other types of normalization, for example normalization over a Layer, Instance, or Group

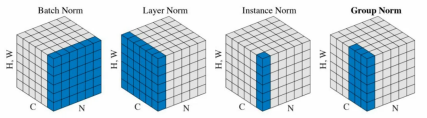


Figure 13: Types of normalization over images (each canal stored over z axis)

When using pre-trained models, it is important to import and use its pre-processing function

4.8 Model Evaluation

When a categorical model is already trained, we could want to evaluate its performance. We can directly compare how the model classified the instances, and what labels they really had. Then, the possible cases could be aggregated:

Definition 4.8: TP

True positive: Given a certain fix class. Correspond to the amount of instances that the model classified as the chosen class, and are actually the chosen class

Definition 4.9: TN

True negative: Given a certain fix class. Correspond to the amount of instances that the model classified as different as the chosen class, and are actually different to the chosen class

Definition 4.10: FP

False positive: Given a certain fix class. Correspond to the amount of instances that the model classified as the chosen class, and are actually different to the chosen class

Definition 4.11: FN

False negative: Given a certain fix class. Correspond to the amount of instances that the model classified as different to the chosen class, and are actually the chosen class

Based on this, it is created the **Confusion Matrix**, that puts these cases, and arranges them conveniently in a matrix:

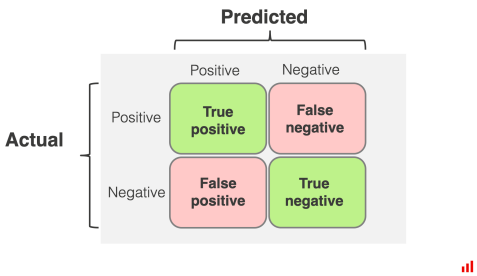


Figure 14: Confusion Matrix with binary classification

Therefore, with this, there are some metrics that we could calculate in order to summarize the model accomplishment in the binary case.

Definition 4.12: Accuracy in binary classification

$\frac{TP+TN}{TP+TN+FP+FN}$

Definition 4.13: Precision in binary classification

$\frac{TP}{TP+FP}$

Definition 4.14: Recall in binary classification

$\frac{TP}{TP+FN}$

Definition 4.15: F1 Score in binary classification

$2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$

In the case of multi-class classification:

Definition 4.16: Accuracy in multi-class classification

$\sum_{i=1}^{N} \frac{TP_i}{TP_i + TN_i + FP_i + FN_i}$, where N is the number of classes

Definition 4.17: Precision$_i$ in multi-class classification

$\frac{TP_i}{TP_i + FP_i}$, where $1 \leq i \leq N$ = the number of classes

Definition 4.18: Recall$_i$ in multi-class classification

$\frac{TP_i}{TP_i + FN_i}$, where $1 \leq i \leq N$ = the number of classes

Definition 4.19: F1$_i$ in multi-class classification

$2 \cdot \frac{precision_i \cdot recall_i}{precision_i + recall_i}$, where $1 \leq i \leq N$ = the number of classes

From this metric, we can create aggregated metrics:

Definition 4.20: Precision$_{macro}$ in multi-class classification

$\frac{1}{N} \sum_{i=1}^{N} Precision_i$

Definition 4.21: Recall$_{macro}$ in multi-class classification

$\frac{1}{N} \sum_{i=1}^{N} Recall_i$

Definition 4.22: F1$_{macro}$ in multi-class classification

$\frac{1}{N} \sum_{i=1}^{N} F1_i$

Or in case we want to give more importance to classes with more instances:

Definition 4.23: Precision$_{weighted}$ in multi-class classification

$\sum_{i=1}^{N} \left(\frac{N_i}{N} Precision_i \right)$

Definition 4.24: Recall$_{weighted}$ in multi-class classification

$\sum_{i=1}^{N} \left(\frac{N_i}{N} Recall_i\right)$

Definition 4.25: F1$_{weighted}$ in multi-class classification

$\sum_{i=1}^{N} \left(\frac{N_i}{N} F1_i\right)$

5 Implementing ANN

There are several ways in what we can implement FFNN's. One of the most used, and simple ones is through python and tensorflow.
This section has been created based on Prof. Eugenio Lomurno's notebook 1 and notebook 2

5.1 Imports and pre-configuration

There are a lots of non-essential parameters that can be modified in order to have a better experience in running the FFNN's. Things like logs and warning raises management could help us in different parts of the coding. We are not going to dive into that
The basic configuration, that specifically we will use consist of:

```
import random # Generating random numbers
############ Manage of data and plots
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
############ Example dataset used
from sklearn.datasets import load_iris
############ Tools for evaluation
from sklearn.metrics import accuracy_score, precision_score, recall_score,
    f1_score, confusion_matrix
############ Cross Validation tool
from sklearn.model_selection import train_test_split

############ ANN library
import tensorflow as tf
from tensorflow import keras as tfk
from tensorflow.keras import layers as tfkl
print(f"TensorFlow version {tf.__version__}")

seed = 69 # It is important fix randomness, so each time we run the code, we
    are going to obtain the same result. This, with debugging and learning
    purposes
np.random.seed(seed)
random.seed(seed)
tf.random.set_seed(seed)
```

5.2 Loading and data inspection

Now we will do the data load and inspection. This process is different for each kind of dataset. In this case, we'll do it with Iris, a classic dataset from sklearn

```
############ Load the Iris dataset into a variable called 'data'
data = load_iris()
############ Print the description of the Iris dataset
print(data.DESCR)
############ Create a DataFrame 'iris_dataset' from the Iris dataset
iris_dataset = pd.DataFrame(data.data, columns=data.feature_names)
print('Iris dataset shape', iris_dataset.shape)
############ Display the first 5 rows of the Iris dataset
iris_dataset.head()
############ Print the shape of the Iris dataset
print('Iris dataset shape', iris_dataset.shape)
############ Generate summary statistics for the Iris dataset
iris_dataset.describe()
############ Get the target values from the Iris dataset
target = data.target
print('Target shape', target.shape)
############ Calculate the unique target labels and their counts
unique, count = np.unique(target, return_counts=True)
print('Target labels:', unique)
for u in unique:
    print(f'Class {unique[u]} has {count[u]} samples')
############ Copy the iris dataset
plot_dataset = iris_dataset.copy()
############ Assign target labels to the dataset
plot_dataset["Species"] = target
############ Plot using seaborn pairplot
sns.pairplot(plot_dataset, hue="Species", palette="tab10", markers=["o", "s",
     "D"])
plt.show()
############ Clean up by deleting the temporary dataset
del plot_dataset
```

5.3 Cross Validation

```
############ Split the dataset into a combined training and validation set,
    and a separate test set
X_train_val, X_test, y_train_val, y_test = train_test_split(
    iris_dataset,
    target,
    test_size=20,
    random_state=seed,
    stratify=target
)
############ Further split the combined training and validation set into a
    training set and a validation set
X_train, X_val, y_train, y_val = train_test_split(
    X_train_val,
    y_train_val,
    test_size=20,
    random_state=seed,
    stratify=y_train_val
```

```
)
############ Print the shapes of the resulting sets
print('Training set shape:\t', X_train.shape, y_train.shape)
print('Validation set shape:\t', X_val.shape, y_val.shape)
print('Test set shape:\t\t', X_test.shape, y_test.shape)
```

5.4 Data Adjustments

```
############ We do normalization, in this case, we do min-max normalization,
    which is different to the one we saw in classes, the mean normalization
max_df = X_train.max()
min_df = X_train.min()

X_train = (X_train - min_df) / (max_df - min_df)
X_val = (X_val - min_df) / (max_df - min_df)
X_test = (X_test - min_df) / (max_df - min_df)

############ Apply one-hot encoding to each data
y_train = tfk.utils.to_categorical(y_train, num_classes=len(unique))
y_val = tfk.utils.to_categorical(y_val, num_classes=len(unique))
y_test = tfk.utils.to_categorical(y_test, num_classes=len(unique))
############ Equivalent to y_train = pd.get_dummies(y_train), taking care
    that every category is in every partition of the data

############ Display shapes of the encoded label sets
print('Training set target shape:\t', y_train.shape)
print('Validation set target shape:\t', y_val.shape)
print('Test set target shape:\t\t', y_test.shape)
```

5.5 Building FFNN

```
batch_size = 16
epochs = 500 # Number of epochs: times the entire dataset is passed through
    the network during training
learning_rate = 0.001 # Learning rate: step size for updating the model's
    weights

def build_model( input_shape=input_shape, output_shape=output_shape,
    learning_rate=learning_rate, seed=seed):
    ############ Fix randomness, just in case
    tf.random.set_seed(seed)

    ############ Build the neural network layer by layer
    inputs = tfkl.Input(shape=input_shape, name='Input')

    ############ Add hidden layer with ReLU activation
    x = tfkl.Dense(units=16, name='Hidden')(inputs)
    x = tfkl.Activation('relu', name='HiddenActivation')(x)

    ############ Add output layer with softmax activation
    x = tfkl.Dense(units=output_shape, name='Output')(x)
    outputs = tfkl.Activation('softmax', name='Softmax')(x)

    ############ Connect input and output through the Model class
```

```
    model = tfk.Model(inputs=inputs, outputs=outputs, name='
    FeedforwardNeuralNetwork')

    ############ Compile the model with loss, optimizer, and metrics
    loss = tfk.losses.CategoricalCrossentropy()
    optimizer = tfk.optimizers.Adam(learning_rate)
    metrics = ['accuracy'] #*
    model.compile(loss=loss, optimizer=optimizer, metrics=metrics)

    ############ Return the model
    return model


############ Build the model with specified input and output shapes
model = build_model()

############ Display a summary of the model architecture
model.summary(expand_nested=True, show_trainable=True)

############ Plot the model architecture
tfk.utils.plot_model(model, expand_nested=True, show_trainable=True,
    show_shapes=True, dpi=70)
```

**: Other metrics can be used, see 5.12*

5.6 Model Train

```
############ Train the model and store the training history
history = model.fit(
    x=X_train,
    y=y_train,
    batch_size=batch_size,
    epochs=epochs,
    validation_data=(X_val, y_val)
).history

############ Calculate the final validation accuracy
final_val_accuracy = round(history['val_accuracy'][-1] * 100, 2)

############ Save the trained model to a file with the accuracy included in
    the filename
model_filename = f'Iris_Feedforward_{final_val_accuracy}.keras'
model.save(model_filename)

############ Delete the model to free up memory resources
del model
```

5.7 Show up training process

We can show up training process compared with the loss, both in training and test set

```
############ Create a figure with two vertically stacked subplots
fig, (ax1, ax2) = plt.subplots(nrows=2, ncols=1, figsize=(15, 6), sharex=True
    )

############ Plot training and validation loss
```

```
ax1.plot(history['loss'], label='Training loss', alpha=.8)
ax1.plot(history['val_loss'], label='Validation loss', alpha=.8)
ax1.set_title('Loss')
ax1.legend()
ax1.grid(alpha=.3)

############ Plot training and validation accuracy
ax2.plot(history['accuracy'], label='Training accuracy', alpha=.8)
ax2.plot(history['val_accuracy'], label='Validation accuracy', alpha=.8)
ax2.set_title('Accuracy')
ax2.legend()
ax2.grid(alpha=.3)

############ Adjust the layout and display the plot
plt.tight_layout()
plt.subplots_adjust(right=0.85)
plt.show()
```

5.8 Early Stopping and CallBacks

If we see an overfitting in our model, based on the loss worsening in test data, we can apply the early stopping technique, seen in section 4.3

```
############ Define the patience value for early stopping
patience = 100 # Example

############ Create an EarlyStopping callback
early_stopping = tfk.callbacks.EarlyStopping(
    monitor='val_mse',
    mode='min',
    patience=patience,
    restore_best_weights=True
)

############ Store the callback in a list
callbacks = [early_stopping]

es_history = model.fit(
    x = X_train,
    y = y_train,
    validation_data = (X_val, y_val),
    batch_size = batch_size,
    epochs = epochs,
    callbacks = callbacks
).history

# Calculate the final validation mse
final_val_mse = round(es_history['val_mse'][-(patience+1)], 4)
print(final_val_mse)

# Save the trained model to a file with the mse included in the filename
model_filename = f'Feedforward_{final_val_mse}.keras'
model.save(model_filename)

# Delete the model to free up memory resources
del model
```

Early Stopping is just one of an extensive list of callbacks, that are tools that allow us to intervene in the training process in determined parts, such as the beginning or end of an epoch.
Another very used is Reduce LR On Plateau, which can reduce the learning rate when the metric has stopped improving. It is used as follows:

```
    reduce_lr_cb = tf.keras.callbacks.ReduceLROnPlateau(
        monitor='val_loss',
        factor=0.5,          # Reduce learning rate by a factor of 0.5
        patience=3           # Wait for 3 epochs before reducing
    )
```

We can create our own callbacks as well, inheriting from 'tf.keras.callbacks.Callback', as follows:

```
class PersonalizedClass(tf.keras.callbacks.Callback):
    def __init__(self, frequency=5, params,...):
        super().__init__()
        self.frequency = frequency # Parameter often used for executing the
    callback only each certain epochs
        # Code
    def on_batch_begin(self, epoch, logs=None):
        # This is executed at the beginning of a training batch.
        # Code to execute
```

There are bunches of other moments in which the callback could be executed. A list of some of them is below:

- **on_batch_end:** Executed at the end of a training batch
- **on_epoch_begin:** Executed at the start of an epoch
- **on_epoch_end:** Executed at the end of an epoch
- **on_predict_batch_begin:** Executed at the start of a batch in predict method
- **on_predict_batch_end:** Executed at the end of a batch in predict method
- **on_predict_begin:** Executed at the start of the predict method. It does not receive batch parameter since it's executed once **on_predict_end:** Executed at the end of the predict method. It does not receive batch parameter since it's executed once
- **on_test_batch_begin:** Executed at the beginning of a batch in evaluate method and the validation batch
- **on_test_batch_end:** Executed at the end of a batch in evaluate method and the validation batch
- **on_test_begin:** Executed at the beginning of evaluation or validation methods. It does not receive batch parameter since it's executed once **on_test_end:** Executed at the end of evaluation or validation methods. It does not receive batch parameter since it's executed once
- **on_train_begin:** Executed at the beginning of training. It does not receive batch parameter since it's executed once

- **on_train_begin:** Executed at the end of training. It does not receive batch parameter since it's executed once

5.9 Weight Decay or Ridge Regression

If we want to apply weight decay or ridge regression (see section 4.4) we have to add this code at the end of the model build process (but before the output activation layer)

```
l2_lambda = 5e-4 # We have to choose the constant

initialiser = tfk.initializers.GlorotNormal(seed=seed)
regulariser = tfk.regularizers.l2(l2_lambda)
output_layer = tfkl.Dense(units=1, kernel_initializer=initialiser,
    kernel_regularizer=regulariser, name='Output')(x)
```

5.10 Dropout

If we want to apply the dropout technique (see section 4.5), we have to add this code after the layer we want to apply the dropout.

```
x = tfkl.Dropout(dropout_rate, seed=seed, name='Dropout')(x)
```

5.11 Inference over new data

```
############ Load the saved model
model = tfk.models.load_model('Iris_Feedforward_95.0.keras')

############ Display a summary of the model architecture
model.summary(expand_nested=True, show_trainable=True)

############ Plot the model architecture
tfk.utils.plot_model(model, expand_nested=True, show_trainable=True,
    show_shapes=True, dpi=70)


############ Predict class probabilities and get predicted classes
val_predictions = model.predict(X_val, verbose=0)
val_predictions = np.argmax(val_predictions, axis=-1)

############ Extract ground truth classes
val_gt = np.argmax(y_val, axis=-1)

############ Calculate and display validation set accuracy
val_accuracy = accuracy_score(val_gt, val_predictions)
print(f'Accuracy score over the validation set: {round(val_accuracy, 4)}')

############ Calculate and display validation set precision
val_precision = precision_score(val_gt, val_predictions, average='weighted')
print(f'Precision score over the validation set: {round(val_precision, 4)}')

############ Calculate and display validation set recall
val_recall = recall_score(val_gt, val_predictions, average='weighted')
print(f'Recall score over the validation set: {round(val_recall, 4)}')

```

```
############ Calculate and display validation set F1 score
val_f1 = f1_score(val_gt, val_predictions, average='weighted')
print(f'F1 score over the validation set: {round(val_f1, 4)}')

############ Compute the confusion matrix
cm = confusion_matrix(val_gt, val_predictions)

############ Create labels combining confusion matrix values
labels = np.array([f"{num}" for num in cm.flatten()]).reshape(cm.shape)

############ Plot the confusion matrix with class labels
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=labels, fmt='', xticklabels=['Setosa', 'Versicolor', '
    Virginica'], yticklabels=['Setosa', 'Versicolor', 'Virginica'], cmap='
    Blues')
plt.xlabel('True labels')
plt.ylabel('Predicted labels')
plt.show()
```

5.12 Metrics

There are a lot of metrics that can be used for training models, such as AUC, accuracy, hinge, IoU, Mean, Precision, Recall, SparseCategoricalCrossentropy, mse, among others. We can custom as well our metrics, as the example in section 17.3.

6 Convolutional Neural Networks

6.1 Introduction

Computer Vision is an interdisciplinary scientific field that deals with how computers can be made to gain high-level understanding from digital images or videos.

Images are made from different canals, that are different levels of information, represented in different matrices. Very often RGB encoding is used, which consists of 3 canals, Red, Green, and Blue. Each canal is one matrix of 8 bits, with values from 0 to 255. On the other hand, videos are sequences of images, placed in order, each one being a frame.

Image classification, as well as regular classification, is treated with a softmax activation output layer. Then, the output layer will be the probability of the image of being part of a certain category $i \in N$, with N predefined.

The approach to feed the Neural Network will be the same, but adding convolutional layers (see section 6.4). It is also possible to use the conventional kind of NN, but it generally gives worse results.

Image Classification has several particular challenging characteristics:

- High dimensionality of images
- Multi-Label images

- Changes in illumination conditions and position of the elements in the photo
- Perceptual Similarity is not necessarily related to Pixel Similarity
- Background changes
- Inter-class variability

A Convolutional Neural Network is a Neural Network that is first passed through Convolutional and Padding Layers.
Then the NN uses the result to run a conventional NN, the dense layers. In this sense, the first part of the CNN could be seen as a feature extraction from the image, to prepare it for the Dense Layers. Applying dense layers requires too much computation since it uses a lot of weights. Decreasing this stage as much as possible allows the NN to perform much better.

The usual structure for this kind of architecture will look like figures 15 and 16.

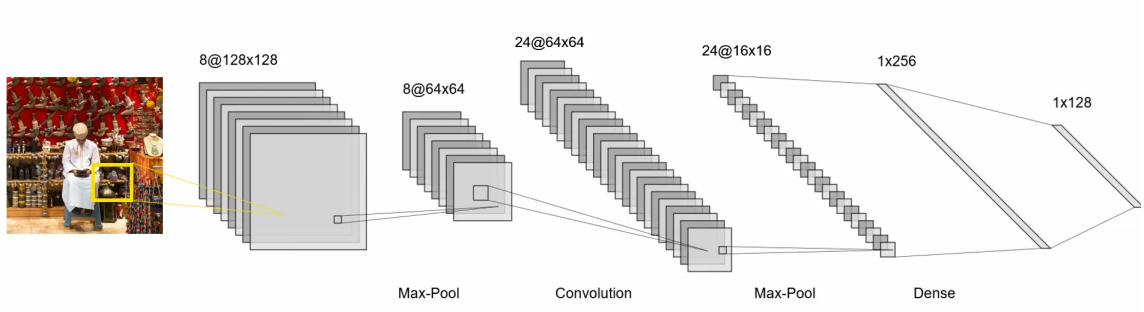


Figure 15: Usual basic structure for CNN's

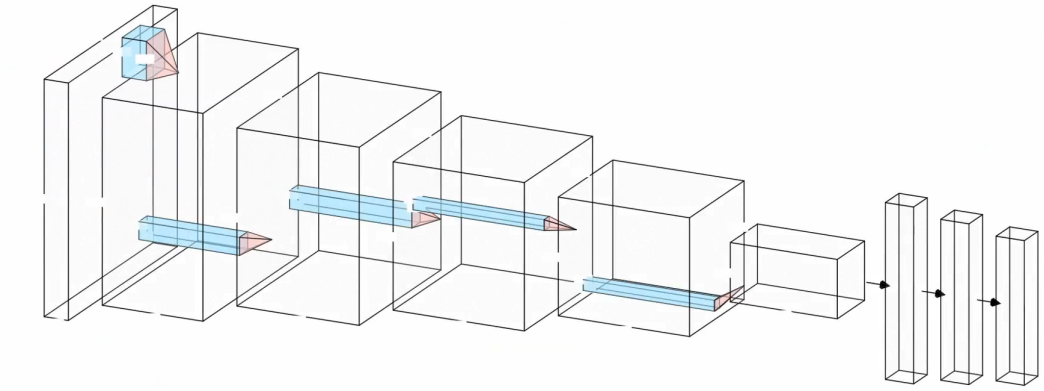


Figure 16: Usual basic structure for CNN's and how the information is passed through layers

6.2 Convolution

Convolution is an operation that concentrates values of a neighborhood of a fixed pixel in the image into one value.

The idea consists of the sum of all the values in the neighborhood of the pixel (r, c) of the image I, weighted by a matrix w:

$$T[I](r,c) = \sum_{(u,v)\in m\cdot U} w(u,v)\cdot I(r+u, c+v)$$

where:

- $U = \{(-1,-1),(-1,0),(-1,1),(0,1),(1,1),(1,0),(1,-1),(0,-1)\}$
- $m \cdot U = \{n \cdot U \ tq \ n \in \{1,\ldots,m\}\}$

$T[I](r,c)$ is a linear combination of the pixels in U
The value of the convolution will be higher while more similar the w matrix is to the neighborhood of (r,c).

6.3 Padding and Stride

When a convolution is needed close to the image boundaries, certain values cannot be calculated with the classical approach. Therefore, when needed, padding (fulfill) the matrix outside the boundary with zeros is the most frequent option.
There are 3 types of padding (see figure 17 as well):

- **No Padding / "valid":** This kind of padding (or the lack of it), implies that convolution will be applied only where it can be calculated with the original image. This makes the output smaller than the input.
- **Half padding / "same":** This kind of padding consists of only adding the zeros needed to obtain the same size output when the stride is 1.
- **Full padding / "full":** This kind of padding consists of adding the maximum zeros such that the filter catches a pixel of the original image. This makes the output bigger than the input

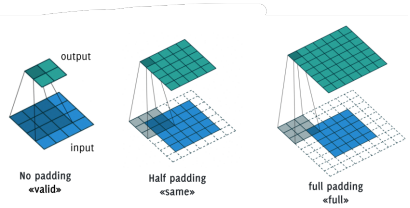


Figure 17: Three types of padding

Besides that, a stride must also be defined, which consists of how many pixels must be skipped to apply the function again. In other words, how many pixels 'the filter will move' to apply the operation again. The default stride in convolutions is 1

6.4 Convolutional Layer

A Convolutional Layer consists of passing the input through some convolutions, plus a bias b, being one bias per convolution applied.

One convolutional layer will be the sequence of convolutions applied including the bias.

Overall, the operation will look like the following expression

$$a(r, c, \ell) = \sum_{(u,v)\in U, k\in \mathbf{C}} [w^{\ell}(u, v, k) \cdot X(r + u, c + v, k)] + b^{\ell}$$

where we add the following notation:

- $\ell \in n_f$
- n_f : Number of desired filters in the convolutional layer.
- $\mathbf{C}$: Number of channels present in the input image
- U will be like $m \cdot U$, but now, we can apply different scalars h_c and h_r in vertical and horizontal axes

Note that the sum of parameters in an image of $h_c \times h_r$ on a convolutional layer will be

$$(h_c \cdot h_r \cdot \mathbf{C} + 1) \cdot n_f$$

In practice, w's and b's will be learned as normal parameters in a FFNN. The usual structure for this kind of architecture are represented in the examples of figures 18 and 19.

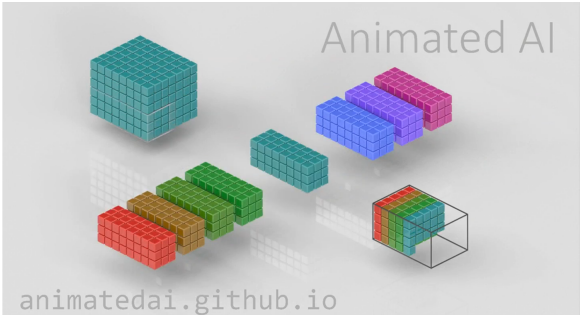


Figure 18: Convolutional Layer, animation frame

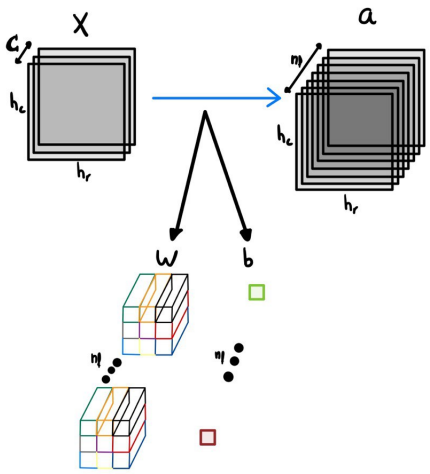


Figure 19: Convolutional Layer dimensions represented

Finally, as well as linear classifiers, very often non-linearities are applied at the end of the layer. Anyway, an activation layer is applied (see section 2.3) in each filter that the convolutional layer created.

6.5 Pooling Layers

Pooling Layers reduce the spatial size (height $\times$ width) of the channels in the image.
The pooling Layer operates independently on every canal of the input and resizes it spatially. It consists of applying a function between $n \times n$ pixels so it transforms them to one (see figure 20). The usual functions applied are max, min, and average. Plus, the default stride is equal to the filter size
A pooling with 2×2 filters and stride 2 reduces the image by a 75%, as we see in the figure 20. On the other hand, if we set a stride 1, it would not reduce the spatial size but just perform pooling on each pixel.
Using a stride=1 makes sense with nonlinear pooling only

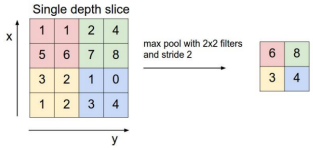


Figure 20: Max pooling example

In some pooling operations like max, the gradient is only routed through the input pixel that contributes to the output value, being 1 if it is the maximum and 0 in another case

6.6 Dense Layers

Finally, the results of the previous layers are passed through a Dense Layer, that are the layers that were seen in the section 2.

6.7 Relation between MLP and CNN

Since the convolution is a linear operation, the operator could be seen as a dense layer. In particular, both convolution and Dense Layers can be described as follows $a = Wx + b$. Actually, the convolution is a specific type of dense layer, adding constraints to the possible weights, taking advantage of the kind of data we are dealing with, images.

Dense layers are already represented as a matrix-vector operation. We can represent convolutional layers with the same notation. Following the section 6.4 notation:

- The input vector will have a size of $h_c \cdot h_r \cdot \mathbf{C}$, that is, large×width×canals.
- The weight matrix will have this same size in the columns, and $h_c \cdot h_r \cdot n_f$, that is large×width×number of filters. This is because each pixel in a certain canal has a weight in the final image.
- The bias will have a size of $h_c \cdot h_r \cdot n_f$, for the same reasons as w.

As we can see in the example of the figure 21, the resulting matrix:

- **Sparse:** It has a lot of 0 values. This is because most of the input pixels don't affect at all the result of the far ones.
- **Shared weights:** Values are repeated in the matrix. This is because the same weights we use for calculating the output in some pixels, for a certain filter, are the same.

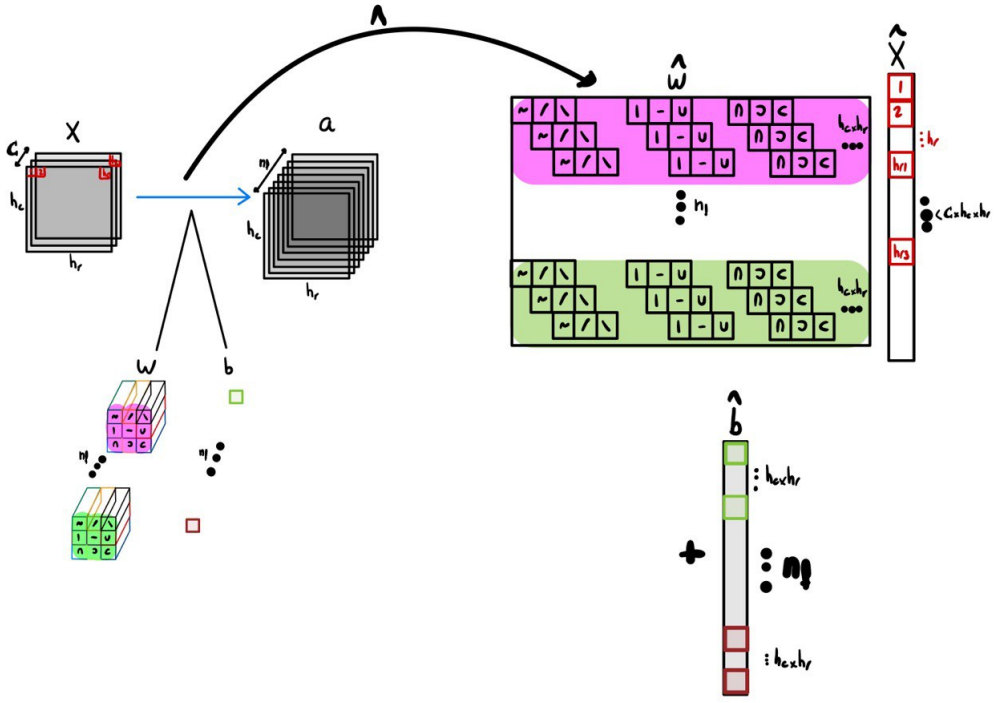


Figure 21: Convolutional Layer represented as a dense layer. Representative, it does not consider boundary image cases

This structure allows us to use much less computation in feed-forward and back-propagation. Knowing their structure, both Sparse and Shared Weights Matrices have properties that can improve execution.

6.8 Receptive Field

Definition 6.1: Receptive Field

Which portion is considered in one output unit for the result, with respect to the initial image

Despite the next layer of CNN very often considers only a portion of the previous one in the image, in turn, that portion of the image is a product of a bigger part of the past one. In this sense, in spite of sparse connectivity, deep CNN can have a very big receptive field in the output. See comparison between figures 22 and 23

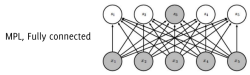


Figure 22: Receptive field of fully connected layer

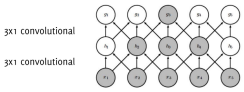


Figure 23: Receptive field of Convolutional Layer

The receptive field could be calculated with heuristics as shown in figure 24

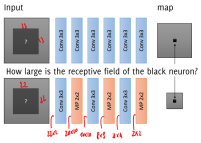


Figure 24: Receptive field heuristic calculations

It is important to note that when calculating the receptive field, this cannot be bigger than image input, so border cases could be problematic in case of having a heuristic to calculate it

In CNN, as we move to deeper layers, spatial resolution is reduced, and the number of maps increases. As we search for higher-level patterns, we don't care too much about their exact location. There are more high-level patterns than low-level details.

7 Lack of Data

Deep Learning models are very data-hungry. They improve their performance while we continue adding data. For this reason, having few data can result in poor results.

7.1 Data Augmentation

Very often, each annotated image represents a class of images that are likely to belong to the same class. In aerial photos, for instance, it is normal to have rotated, shifted, or scaled images without changing the label.

A technique for taking advantage of this is called data augmentation. It is usually performed by geometric or photometric transformations, categories with different types of transformation.

- Geometric Transformations:
  - Shift, rotations, affine, perspective distortions
  - Shear
  - Scaling
  - Flip
- Photometric Transformations:
  - Adding noise
  - Modifying average intensity
  - Superimposing other images
  - Modifying image contrast

Augmented versions must preserve the input label. For example, if size is a key information to determine the output target, scaling would not be a proper transformation for augmentation. There are other cases where it is possible to apply augmentation, but it must include a transformation to the label. This is common in image segmentation (see section 10)

We have to be careful with applying transformations only to certain classes, because the model could learn class-discriminative patterns that were not inside the non-augmented dataset.

7.1.1 Mixup Augmentation

A special type of augmentation is the Mixup Augmentation, which consists of superimposing images, creating a new one, taking the one hot encoded label, and defining its coordinates based on the transparency of the images, giving it the correspondent probability to the new sample output.

7.1.2 Test Time augmentation or self-ensembling

Even if the CNN is trained using augmentation, it won't achieve perfect invariance. This is why augmentation could be used in the test set as well. It consists of performing a few random augmentation of each test image. Then, we classify the original and augmented images. The CNN result will be the prediction by aggregating the posterior vectors of all images, for instance, doing an average.

7.2 Transfer Learning and fine tuning

There are bunches of models already trained that perform very well in their tasks.

As we see before, the convolutional part of the model is thought as the feature extractor part of the model. Then, the part of the model in charge of actually calculating the class of an image is the dense layer, the last one.

Then, if we have a few images for a different task than the original one, it is possible to remove and retrain dense layers of the model. This is called **Transfer Learning**. This approach would keep a significant part of the model fixed, the feature extraction, using this for performing the classifier over the dense layers.

Another option is to retrain the whole CNN, but the convolutional layers are initialized as the pre-trained model. This is called **Fine Tuning**. This is a good option when we have a little more amount of data. Fine-tuning performs well when Transfer Learning was applied before, so it must be used only after it.

Typically, when the optimizer is the same, smaller learning rates are used to perform these methods

One of the most known powerful trained CNN are:

- ResNet
- EfficientNet
- MobileNet

8 History of famous CNN: Architectures and New Layers

In recent years, some important models have been released, especially in the context of the classification competencies as Imagenet

8.1 LeNet-5

LeNet-5 is the first CNN of history. Was developed in 1998 by Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. It consisted of 2 convolutional and average-pooling layers, followed by 3 dense layers (See figure 25). It was designed for document recognition.

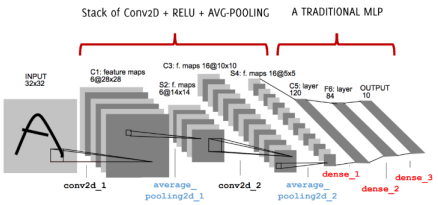


Figure 25: LeNet-5 architecture

8.2 AlexNet

AlexNet was developed in 2012 by Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. It consisted of 5 convolutional layers with rather large filters (11x11 and 5x5) and 3 MLP. It had 60 million of parameters, where 94% where part of fully connected layers. It also introduced RELU, dropout, weight decay, norm layers, and maxpooling (See figure 26).

They used this architecture for two volumes over different GPUs, mixing them only by a few connections. In the end, they also trained an ensemble of 7 models to drop error: 18,2% to 15.4% It was designed for document recognition.

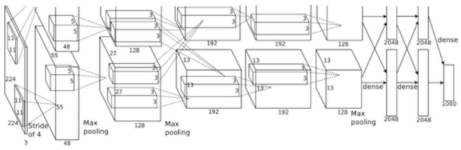


Figure 26: AlexNet architecture

8.3 VGG16

VGG16 was developed in 2014 by Karen Simonyan and Andrew Zisserman. It consisted of a deeper variant of the AlexNet structure. Smaller filters were used (3x3) and the network was deeper. This allows them to reduce the number of parameters and have more nonlinearities (having a lot of convolutional layers without pooling, making blocks of convolutions). Even though, the number of parameters grew more than twice, with 138 million, where 89% were part of fully connected layers (See figure 27), this was possible because of the improvements in computational power. This architecture won the first place in localization and the second place in classification in ImageNet. It required high memory, about 100MB per image only in the forward pass, and about 200MB during the backward

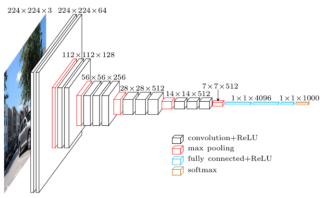


Figure 27: VGG16 architecture

8.4 Networks in Networks

Networks in Networks was developed in 2014 by Min Lin, Qiang Chen, and Shuicheng Yan. It consisted of, instead of convolutional layers, using a sequence of FC + RELU. It used a stack of FC layers followed by RELU in a sliding manner on the entire image. It corresponds to MLP networks used convolutionally as it can be see in figure 28. As a result, each layer features had a more powerful functional approximation than a convolutional layer that is just linear + RELU.

This paper also introduced the Global Averaging Pooling (GAP) Layer. It consisted of

calculating the average over all the pixels in a whole channel (through 512 channels), reducing the dimensionality of the problem rapidly. It could be represented as multiplication against a block diagonal constant matrix. They were developed since FC layers are prone to overfitting, due to many parameters. GAP replaced the fully connected layers, doing a soft-max just after it.

As GAP does not have parameters, this structure is much lighter than the previous ones, being less prone to overfitting and easier to run. It is also invariant to shifts of the input image, being more powerful in identifying translated images

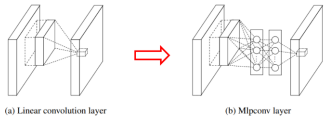


Figure 28: Networks in Networks architecture

The network is used to classify images of different sizes

8.5 InceptionNet V1 / GoogLeNet

InceptionNet V1, also known as GoogLeNet, was developed in 2014 by Christian Szegedy et.al. It improved the performance of NN's by increasing their size, but taking care of the problem that a large number of parameters, makes the enlarged network more prone to overfitting. Moreover, image features might appear a different scales, then it is difficult to set the right kernel size.

What InceptionNet did was run convolutions (and MaxPooling) with different kernel sizes (3x3 and 5x5) in parallel, and then concatenate them in a concatenation filter, making sure that all the runs have the same output extent (this is done with stride and padding). The problem with just doing this is that the concatenated output increases its channel number a lot concerning the input.

To solve the increasing size, 1x1 convolutions are performed before the convolution (and after max pooling). This decreases the number of channels, as shown in figure 29, with a not-expensive operation, since the number of filters can be chosen. Plus, it increases the number of non-linearity.
This portion of the structure is as shown in figure 30

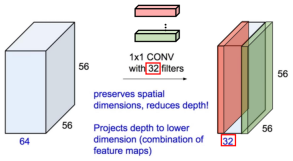


Figure 29: Reduction in size of images with 1x1 convolutions

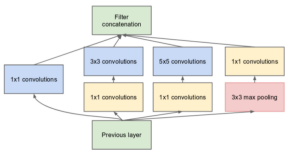


Figure 30: InceptionNet zoomed architecture

In a general view, GoogleNet stacked 27 layers considering pooling ones. Plus, at the beginning, there are two blocks of normal convolutional and pool layers. In the end, there was no FC layer, just a simple GAP, linear classifier, and a softmax. This whole structure, used 5 million of parameters.

As a last aggregate, the authors solved the dying neuron problem by adding two extra auxiliary classifiers (GAP+linear classifier+softmax) on the intermediate representation to compute an intermediate loss that is used during the training. This gave the model the capacity to provide meaningful features for classification as well. At inference time, these extra classifiers were removed. The general structure can be seen in figure 31

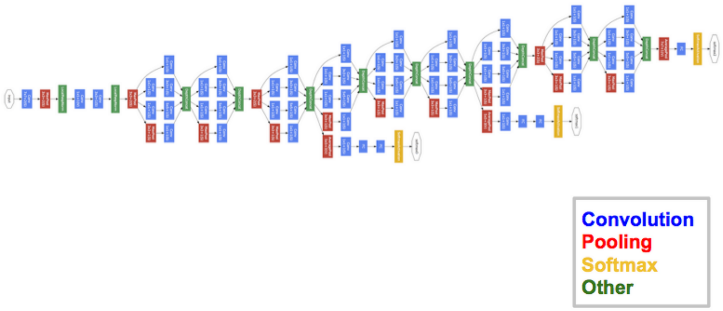


Figure 31: InceptionNet whole architecture

Overall, InceptionNet provided us with multiple improvements:

1. Blocks made of multiple connections instead of having single treads work better
2. In order to reduce bottlenecks, 1x1 convolution is a good practical tool. It helps to reduce the number of operations and parameters of the network
3. Train the network on additional tasks can improve training convergence, avoiding vanishing gradient descent

8.6 ResNet

ResNet was developed in 2015 by Microsoft Research. It increases the number of layers to 152 in Imagenet and 1202 in CIFAR. It was ILDVR winner in both localization (see section 11) and classification. It achieved 3.57% in classification error, better than human performance. The main investigation was about how could be possible to continuously improve accuracy by stacking layers.

Before ResNet, it was noted that despite more layers should lead to a better error, there was a point where more layers in a CNN led to more error. This didn't make sense, because the network could have approximated just an identity function in the last part to emulate the CNN with fewer layers. This was not overfitting, because the same trend was shown in training and test data.

Convinced that was because of the architecture, they concluded that the identity function is not easy to learn. Therefore, they created an 'identity shortcut connection', a connection that consisted of creating a new connection that skipped layers with an identity function (see figure 32).

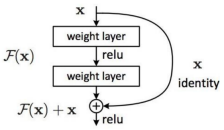


Figure 32: Identity shortcut connection schema

This architecture, called residual block, helped mitigate the vanishing gradient and therefore enabled deeper architectures without adding additional parameters. This allows propagating the information by the identity in the case when weights to be learned go to zero.

If $\mathbf{H}(x)$ is the ideal mapping to be learned from a plain network, we force the network to learn $\mathbf{F}(x) = \mathbf{H}(x) - x$, a term called residual.

ResNet is a stack of 152 layers of this architecture. The network alternates between some spatial pooling by convolution with stride 2, and doubling the number of filters.

8.7 MobileNet

MobileNet was developed in 2017 by Google Inc, achieving reduce dramatically the number of operations done by the network, originally having $D_k \times D_k \times M \times D_F \times D_F \times N$ operations. It did it by developing a new way to do convolution, based on two steps:

1. Depth-wise convolution that does not mix channels. It is exactly like normal 2D convolution but with different weights for every input channel. It makes $D_k \times D_k \times M \times D_F \times D_F$ operations, since only one filter is used for the channel.

2. Point-wise convolution combines the output of depth-wise convolution by N filters that are 1x1. It does not find spatial patterns anymore, it just mixes the results of the uncombined depth-wise convolutions. It makes $M \times D_F \times D_F N$ operations

It could be seen that the ratio between these two operations and the normal convolution is

$$\frac{1}{N} + \frac{1}{D_K^2}$$

Which denotes a substantial computational saving when N and D_K are large

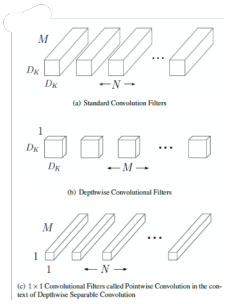


Figure 33: Number of operations comparison between MobileNet and normal convolution

MobileNet achieves lower accuracy than some models before seen, but the performance is not far, and the computational cost saving is crucial.

8.8 Some recent models

In 2017, Facebook developed 2 variants of the ResNet architecture

8.8.1 Wide Resnet

It consisted of using wider residual blocks, F x k filters instead of F filters on each layer. 50-layer Wide Resnet outperforms 152-layer ResNet.

Increasing width instead of depth makes it parallelizable and so more computationally efficient.

8.8.2 ResNeXt

It consisted of using multiple parallel pathways. Similar to the inception module, the activation maps are processed in parallel.

8.8.3 DenseNet

It was developed by Gao Huang et.al. in 2017. It consisted of each convolutional layer taking as input the output of all the previous layers.

There are short connections between the convolutional layers of the network. Each layer is connected to every other layer in a feed-forward fashion.

This alleviates the vanishing gradient problem, promoting feature re-use since each feature is spread through the network

8.8.4 SENet

It was developed by Jie Hu et.al. in 2018. It is the acronym for Squeeze-and-Excitation Networks. It introduced a module that improved the performance by recalibrating each channel for a certain factor.

The mechanism is done by channel-wise attention. It consists of developing a parallel path, starting with a GAP. Then, the exitation part consists of a small dense layer that decides the importance of each channel. Finally, the scale is applied to each channel in a multiplicative way.

8.8.5 EfficientNet

In 2019, Mingxing Tan and Quoc V. Le proposed a new scaling method that uniformly scales all dimensions of depth/width/resolution using a simple but highly effective compound coefficient.

In the figure 34, we observe the difference in size, operations, and accuracy in the models previously described

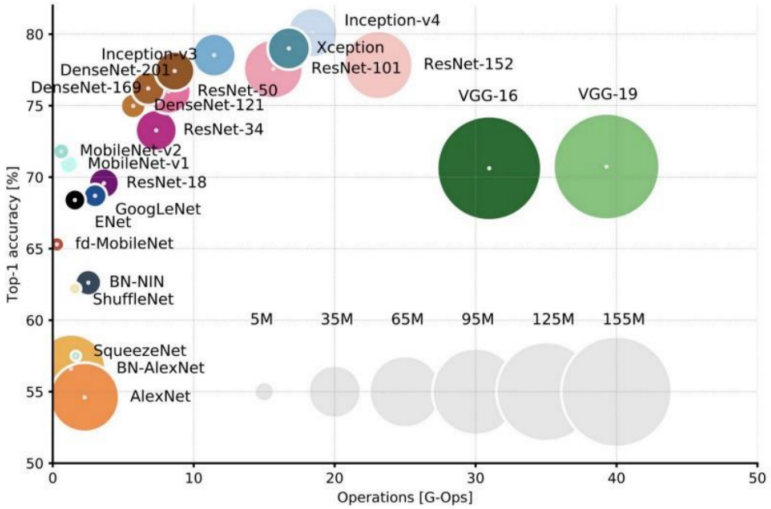


Figure 34: Number of operations comparison between architectures

9 Implementing CNN's

This section has been created based on Prof. Eugenio Lomurno's notebook 1 and notebook 2

9.1 Aditional Imports

```
from PIL import Image
import matplotlib.gridspec as gridspec
```

9.2 Load Data

```
############ Import Data example (cifar10)
(X_train_val, y_train_val), (X_test, y_test) = tfk.datasets.cifar10.load_data
    ()

############ Insert label names
labels = {0:'Airplane', 1:'Automobile', 2:'Bird', 3:'Cat', 4:'Deer', 5:'Dog',
     6:'Frog', 7:'Horse', 8:'Ship', 9:'Truck'}
unique_labels = list(labels.values())
```

9.3 Setting data type

```
############ Normalize data to the range [0, 1], images are in RGB
X_train_val = (X_train_val / 255).astype('float32')
X_test = (X_test / 255).astype('float32')

############ Convert labels to categorical format using one-hot encoding
y_train_val = tfk.utils.to_categorical(y_train_val)
y_test = tfk.utils.to_categorical(y_test)
```

9.4 Augmentation Data

```
input_layer = tfkl.Input(shape=input_shape, name='Input')
############ include augmentation layers
a = tfkl.RandomFlip("horizontal_and_vertical")
b = tfkl.RandomTranslation(0.1,0.1)
c = tfkl.RandomRotation(0.1)
d = tfkl.RandomZoom(0.2)
e = tfkl.RandomBrightness(0.5, value_range=(0,1)) # Here we use values that
    make it obvious, but often less aggressive ones are used
f = tfkl.RandomContrast(0.75)

model = keras.model.Sequential()
model.add(input_layer)
model.add(a)
model.add(b)
model.add(c)
model.add(d)
model.add(e)
model.add(f)
```

These are some of the most common ones, but they shouldn't be applied always. A decision should be taken with respect to the data and its characteristics

Personalized function can be also applied to static data, this is done with a decorator tf.function. See the following example

```
    @tf.function
    def random_flip(image, label, seed=None):
        """Consistent random horizontal flip."""
        if seed is None:
            seed = np.random.randint(0, 1000000)
        flip_prob = tf.random.uniform([], seed=seed)
        image = tf.cond(
            flip_prob > 0.5,
            lambda: tf.image.flip_left_right(image),
            lambda: image
        )
        label = tf.cond(
            flip_prob > 0.5,
            lambda: tf.image.flip_left_right(label),
            lambda: label
        )
        return image, label
```

9.5 Learning part of model

```
conv2 = tfkl.Conv2D(
filters = n_f,
kernel_size = (h_r, h_c),
activation = 'relu',
strides = (1,1),
padding = 'same',
name = 'conv2'
)
act = tfkl.Activation('relu', name='act1')
pool = tfkl.AveragePooling2D(
pool_size=(2, 2)
)
flat = fltfkl.Flatten(name='flatten')
dense = tfkl.Dense(units=output_shape, name='dense')
output = tfkl.Activation('softmax', name='softmax')


model.add(conv2)
model.add(act)
model.add(pool)
model.add(flat)
model.add(dense)
model.add(output)
model.summary()
```

9.6 Inception Block

```

input_layer = model.input
x = model.output

# 1x1 convolutional path
conv1 = tfkl.Conv2D(filters // 4, 1, padding=padding, name=f'{name}_conv1_{s}
    ')(x)
conv1 = tfkl.Activation(activation)(conv1)

# 3x3 convolutional path with initial reduction
conv3_reduce = tfkl.Conv2D(filters // 8, 1, padding=padding)(x)
conv3_reduce = tfkl.Activation(activation)(conv3_reduce)
conv3 = tfkl.Conv2D(filters // 4, 3, padding=padding, name=f'{name}_conv3_{s}
    ')(conv3_reduce)
conv3 = tfkl.Activation(activation)(conv3)

# 5x5 convolutional path with initial reduction
conv5_reduce = tfkl.Conv2D(filters // 12, 1, padding=padding)(x)
conv5_reduce = tfkl.Activation(activation)(conv5_reduce)
conv5 = tfkl.Conv2D(filters // 4, 5, padding=padding, name=f'{name}_conv5_{s}
    ')(conv5_reduce)
conv5 = tfkl.Activation(activation)(conv5)

# Pooling path with projection
pool = tfkl.MaxPooling2D(3, strides=1, padding=padding)(x)
pool_proj = tfkl.Conv2D(filters // 4, 1, padding=padding)(pool)
pool_proj = tfkl.Activation(activation)(pool_proj)
```

```
25
26 # Concatenate paths
27 output = tfkl.Concatenate(name=f'{name}_concat_{s}')([conv1, conv3, conv5,
       pool_proj])
28
29
30 model_with_inception = Model(inputs=input_layer, outputs=output)
31 model_with_inception.summary()
```

We can also perform Batch Normalization, which consists of adding a normalization layer after each convolution. For instance:

```
1 conv = tfkl.BatchNormalization(name=f'{name}_bn_{s}')(conv)
```

9.7 Identity Shortcuts

```
1  input_layer = model.input
2  x = model.output
3
4  s1 = tfkl.Conv2D(
5      filters=filters,
6      kernel_size=3,
7      padding='same',
8      activation='relu',
9      name='conv'+name+'-'+str(1)
10 )(x)
11 s2 = tfkl.Conv2D(
12     filters=filters,
13     kernel_size=3,
14     padding='same',
15     activation='relu',
16     name='conv'+name+'-'+str(c+2)
17 )(s1)
18 s3 = tfkl.Add(name='add'+name)([x,s2])
19 s4 = tfkl.ReLU(name='relu'+name)(s3)
20 s5 = tfkl.MaxPooling2D(name='pooling'+name)(s4)
21
22 model_with_shortcuts = Model(inputs=input_layer, outputs=output)
23 model_with_shortcuts.summary()
```

9.8 SENet

```
1  def senet_block(x, filters, kernel_size=3, padding='same',
2                  downsample=True, activation='relu', stack=2, name='senet'):
3
4      for s in range(stack):
5          # Main convolutional path
6          x = tfkl.Conv2D(filters, kernel_size, padding=padding,
7                          use_bias=False, name=f'{name}_conv_{s}')(x)
8          x = tfkl.BatchNormalization(name=f'{name}_bn_{s}')(x)
9          x = tfkl.Activation(activation, name=f'{name}_act_{s}')(x)
10
11         # Squeeze-and-Excitation (SE) module
12         channels = x.shape[-1]
13
```

```
        # Squeeze step
        se = tfkl.GlobalAveragePooling2D(name=f'{name}_squeeze_{s}')(x)

        # Excitation step
        se = tfkl.Dense(channels // 16, activation=activation, name=f'{name}_dense1_{s}')(se)
        se = tfkl.Dense(channels, activation='sigmoid', name=f'{name}_dense2_{s}')(se)

        # Scaling of the output with SE activation
        se = tfkl.Reshape((1, 1, channels))(se)
        x = tfkl.Multiply(name=f'{name}_scale_{s}')([x, se])

    # Optional downsampling
    if downsample:
        x = tfkl.MaxPooling2D(2, name=f'{name}_pool')(x)

    return x
```

9.9 MobileNet

```
def MobileNetBlock(x, filters, kernel_size, padding, activation='relu', name='mobilenet'):
    #Depthwise convolution
    x = tfkl.DepthwiseConv2D(kernel_size, strides=1, padding=padding, use_bias=False, name=f'{name}_depthwise')(x)
    x = tfkl.BatchNormalization(name=f'{name}_bn1')(x)
    x = tfkl.Activation(activation, name=f'{name}_act1')(x)

    #Pointwise convolution
    x = tfkl.Conv2D(filters, 1, padding=padding, use_bias=False, name=f'{name}_project')(x)
    x = tfkl.BatchNormalization(name=f'{name}_bn2')(x)
    x = tfkl.Activation(activation, name=f'{name}_act2')(x)
    return x
```

9.10 Train Model

The same as it is done in Regular Neural Networks

9.11 Visualize convolution activations

```
############
first_activations, second_activations = extract_activations(model, X, num_images)
fig, axes = plt.subplots(1, 8, figsize=(16, 14))
for i in range(8):
    ax = axes[i]
    ax.imshow(first_activations[image, :, :, i], cmap='bone', vmin=0., vmax=1.)
    ax.axis('off')
    if i == 0:
        ax.set_title('First convolution activations', loc='left')
plt.tight_layout()
```

```
plt.show()
```

9.12 Image Retrieval

This is calculating the most similar images in a dataframe. It is needed a previous processing, since as we have already seen, image distance is not related with image similarity. Specifically, the convolutional part of a CNN is very good for extracting characteristics from images.

```
############ Create an embedding model by removing the dense layer of the
    original model. In this case, it is only one dense layer
embedding = tfk.Sequential(model.layers[:-1])

############ Display the summary of the embedding model architecture
embedding.summary()

############ Extract and preprocess a single image for feature extraction
index = 100
image = np.expand_dims(X[index], axis=0)

############ Predict the features of the selected image using the embedding
    model
image_features = embedding.predict(image, verbose=0)

############ Display the selected image
plt.imshow(X[index])
plt.xticks([])
plt.yticks([])
plt.show()

############ Extract features from the entire dataset using the embedding
    model
dataset_features = embedding.predict(X, batch_size=32, verbose=0)

############ Compute the distances between the selected image's features and
    the entire dataset's features
distances = np.mean(np.square(dataset_features - image_features), axis=-1)

############ Sort images by their distances (similarity to the selected image
    )
ordered_images = X[distances.argsort()]

############ Display the top 10 most similar images
num_img = 10
fig, axes = plt.subplots(1, num_img, figsize=(20, 20))
for i in range(num_img):
    ax = axes[i % num_img]
    ax.imshow(ordered_images[i])
    ax.set_xticks([])
    ax.set_yticks([])
plt.tight_layout()
plt.show()
```

9.13 Use existing models

Use existing models, in this case, we don't use transfer learning or fine-tuning

```
############ Load the MobileNetV2 model pre-trained on ImageNet, without the top classification layer
mobilenet = tfk.applications.MobileNetV2(
    input_shape=(224, 224, 3),
    include_top=False,
    weights="imagenet",
    pooling='avg',
)

############ Display a summary of the model architecture
mobilenet.summary(expand_nested=True, show_trainable=True)

############ Extract and preprocess a single image for feature extraction using MobileNetV2
index = 100
image = np.expand_dims(X[index], axis=0)

############ Preprocess the image and predict its features using the MobileNetV2 model
image_features = mobilenet.predict(preprocess_input(image * 255), verbose=0)
```

9.14 Transfer Learning and Fine-tuning

Transfer Learning

```
############ Initialise MobileNetV3Small model with pretrained weights, for transfer learning
mobilenet = tfk.applications.MobileNetV3Small(
    input_shape=(128, 128, 3),
    include_top=False, # Important: Indicates whether we add dense layers or not, necessary if we want to change the input shape
    weights='imagenet',
    pooling='avg',
    include_preprocessing=True
)
############ Freeze all layers in MobileNetV3Small to use it solely as a feature extractor
mobilenet.trainable = False
############ Define input layer with shape matching the input images
inputs = tfk.Input(shape=(128, 128, 3), name='input_layer')
############ Apply data augmentation for training robustness
augmentation = tf.keras.Sequential([
    tfkl.RandomFlip("horizontal"),
    tfkl.RandomTranslation(0.2, 0.2)
], name='preprocessing')

x = augmentation(inputs)
############ Pass augmented inputs through the MobileNetV3Small feature extractor
x = mobilenet(x)
############ Add a dropout layer for regularisation
x = tfkl.Dropout(0.3, name='dropout')(x)
############ Add final Dense layer for classification with softmax activation
```

```
outputs = tfkl.Dense(y_train.shape[-1], activation='softmax', name='dense')(x)
############ Define the complete model linking input and output
tl_model = tfk.Model(inputs=inputs, outputs=outputs, name='model')
############ Compile the model with categorical cross-entropy loss and Adam optimizer
tl_model.compile(loss=tfk.losses.CategoricalCrossentropy(), optimizer=tfk.optimizers.Adam(), metrics=['accuracy'])
```

Fine-tuning

```
############ Set the MobileNetV3Small model layers as trainable
mobilenet.get_layer('MobileNetV3Small').trainable = True
############ Set all MobileNetV3Small layers as non-trainable
for layer in mobilenet.get_layer('MobileNetV3Small').layers:
    layer.trainable = False
############ Enable training only for Conv2D and DepthwiseConv2D layers
for i, layer in enumerate(mobilenet.get_layer('MobileNetV3Small').layers):
    if isinstance(layer, tf.keras.layers.Conv2D) or isinstance(layer, tf.keras.layers.DepthwiseConv2D):
        layer.trainable = True
############ Set the first N layers as non-trainable
N = 124
for i, layer in enumerate(mobilenet.get_layer('MobileNetV3Small').layers[:N]):
    layer.trainable = False
```

10 Image Segmentation

10.1 The task

Image segmentation aims to identify groups of pixels that 'go together'. This could be through grouping similar-looking pixels or separating images into coherent objects. See the next example

Figure 35: Image segmentation example, SLIC segmentation case

Mathematically, given an image $I \in \mathbf{R}^{R \times C \times 3}$ having as domain $\mathcal{X}$, the goal of image segmentation consists of estimating a partition $\{R_i\}$ such that

$$\bigcup_i R_i = \mathcal{X} \qquad \wedge \qquad R_i \cap R_j = \varnothing, i \neq j$$

Specifically, there are two types of segmentation

1. Unsupervised, which is not addressed in this course
2. Supervised (or semantic segmentation)

10.2 Semantic Segmentation

The specific task is, given an image $I \in \mathbb{R}^{R \times C \times 3}$, associate to each pixel (r,c) a label from Λ, a fixed set of categories. It could be represented as

$$I \rightarrow S \in \Lambda^{R \times C}$$

Where $S(x, y) \in \Lambda$ denotes the class associated to the pixel (x,y). The result is a map of labels containing the estimated class in each pixel.

For this purpose, the training set is a set of images manually annotated

$$TR = \{(I, GT)_i\}$$

Instances are not recognized between them beyond the label identification. In other words, if two objects are part of a class, they are categorized as the same, without any difference.

10.3 Initial Approaches

One approach could consist of

1. Train a classifier on a sufficiently large portion of images (patch)
2. Crop all the patches from the input image to be segmented
3. Classify each patch separately
4. Assign the predicted class to the central pixel of the patch

The problem with this is that it is attached to an arbitrary crop, that does not necessarily match the desired result. Plus, this is a very computationally expensive solution, since a lot of images are obtained

Another approach is using convolution only. It consists of applying only convolutions, avoiding any pooling layer. The last layer should have C canals, used for a softmax function to obtain the required classification for each pixel.
The problem with this is that it is inefficient and each pixel does not have enough receptive field

10.4 Most used approach

We need to 'go deep' to extract high-level information. On the other hand, we don't want to lose spatial resolution in the predictions.

An architecture that meets these goals combines contractive and expanding layers, as figure 36

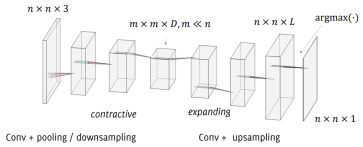


Figure 36: Down and up-sampling convolution architecture

This approach takes two stages. The encoder stage, that corresponds to the downsampling part of the CNN. The decoder stage, that corresponds to the upsampling part.

10.5 Upsampling

There are diverse ways to perform upsampling

10.5.1 Nearest Neighbor

It corresponds to assigning each pixel to its nearest neighbor

10.5.2 "Bed of Nails"

It corresponds to assigning the label to a fixed position and keeping all the other pixels as 0

10.5.3 Max Unpooling

It is like Bed of Nails, but instead of assigning the label to a fixed position, it is assigned to the position where the pixel was extracted in a previous pooling in the encoder stage. This position should have been previously saved. We see an example in figure 37

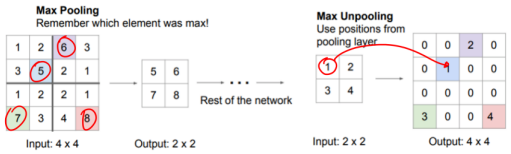


Figure 37: Max-unpooling example

10.5.4 Transposed Convolution

It consists of:

1. Calculating filter matrix scaled for each pixel in the input
2. Summing the results, stacking the next matrix moving it $\langle strides \rangle$ units.

as it is seen in the figure 38, with 3 examples of contractions

$$Input = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

$$Filter = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

$$1 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 2 & 2 & 2 \\ 2 & 2 & 2 \\ 2 & 2 & 2 \end{bmatrix} \begin{bmatrix} 3 & 3 & 3 \\ 3 & 3 & 3 \\ 3 & 3 & 3 \end{bmatrix} \begin{bmatrix} 4 & 4 & 4 \\ 4 & 4 & 4 \\ 4 & 4 & 4 \end{bmatrix}$$

$$2.a = \begin{bmatrix} 1 & 1+2 & 1+2 & 2 \\ 1+3 & 1+2+3+4 & 1+2+3+4 & 2+4 \\ 1+3 & 1+2+3+4 & 1+2+3+4 & 2+4 \\ 3 & 3+4 & 3+4 & 4 \end{bmatrix}$$

$$2.b = \begin{bmatrix} 1 & 3 & 3 & 2 \\ 4 & 10 & 10 & 6 \\ 4 & 10 & 10 & 6 \\ 3 & 7 & 7 & 4 \end{bmatrix}$$

Figure 38: Example of transposed convolution with a 2×2 input, 3×3 input, and stride 1

In practice, padding is applied to better control the output size. In tensorflow, the default output is equal to the input size

10.6 Cross-Entropy loss

The loss function becomes the sum of pixel-wise losses

$$\hat{\theta} = argmin_{x_j \in l} \sum_{x_j \in R} \ell(x_j, \theta; y_j)$$

where

- x_j are all the pixels in a region R of the input Image
- y_j is the actual label in data

10.7 U-Net

U-net is one of the most famous architectures of CNN for image segmentation. It was developed by Olaf Ronneberger, Philipp Fischer, and Thomas Brox in 2015.

It consisted in both a contracting and expansive path with no fully connected layers. It uses a large number of feature channels in the upsampling part, making the network

symmetric. It also uses excessive data augmentation by applying elastic deformations to the training images.

In the contracting path it repeats blocks of 3x3 convolutions + RELU, with valid option (no padding), plus a 2x2 max-pooling. Plus, at each downsampling, the number of feature maps is doubled.

In the expanding path, it repeats blocks of 2x2 transpose convolutions, halving the number of feature maps, but doubling spatial resolution. Plus, it applies twice a 3x3 convolution + RELU.

It has skip connections as well, from the contracting to the expanding part, using only a certain part of the image in downsampling, as seen in figure 39

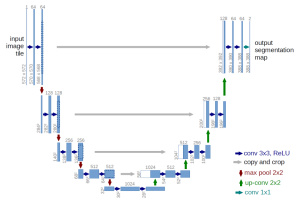


Figure 39: U-net architecture

This architecture has no fully connected layers, only convolutions. Plus, the output is smaller than the image by a constant border.
Finally, it used a weighted loss function.

$$\hat{\theta} = min_\theta \sum_{x_j} w(x_j)\ell(x_j, \theta)$$

where the weight is

$$w(x) = w_c(x) + w_0 e^{-\frac{(d_1(x)+d_2(x))^2}{2\sigma^2}}$$

where

- w_c is used to balance class proportions (remember no patch resampling in full-image training)
- d_1 is the distance to the border of the closest cell
- d_2 is the distance to the border of the second closest cell

The implementation in tensorflow can be done as follows:

```
def unet_block(input_tensor, filters, kernel_size=3, activation='relu'
    , name=''):
    # 2D convolution
    x = tfkl.Conv2D(filters, kernel_size=3, padding='same', name=name+'conv1'
    )(input_tensor)
    # batch normalization (optional)
    x = tfkl.BatchNormalization(name=name+'bn1')(x)
    # activation after batchnorm (this is kind of debated)
    x = tfkl.Activation(activation, name=name+'activation1')(x)
    # 2D convolution
    x = tfkl.Conv2D(filters, kernel_size=3, padding='same', name=name+'conv2'
    )(x)
```

```
    # batch normalization (optional)
    x = tfkl.BatchNormalization(name=name+'bn2')(x)
    # activation
    x = tfkl.Activation(activation, name=name+'activation2')(x)
    return x
```

```
# Forth Downsampling
down_block_4 = unet_block(d3, 256, name='down_block4_')
d4 = tfkl.MaxPooling2D()(down_block_4)
d4 = tfkl.Dropout(0.2, seed=seed)(d4)
# Bottleneck
bottleneck = unet_block(d4, 512)
# First Upsampling layer
# add 2D upsampling layer (this has learnable weights)
u1 = tfkl.UpSampling2D()(bottleneck)
# add dropout
u1 = tfkl.Dropout(0.2, seed=seed)(u1)
# concatenate the output of down_block_4 (the deepest layer in the
    contractive path) wit
h the first layer in the expanding path.
# Spatial sizes need to be consistent, otherwise it is not possible to
    concatenate
u1 = tfkl.Concatenate()([u1,down_block_4])
# add a convolutional block (Exactly same architecture as in the contractive
    path)
u1 = unet_block(u1, 256, name='up_block1_')
```

10.8 Fully Convolutional Neural Networks (FCNN)

The typical architecture of a convolutional neural network consists of convolutional layers followed by dense layers.

The dense layers could be written as convolutional layers as well, as seen in section 6.7.

This is especially useful because convolutional layers work with different input sizes as well. Therefore, representing dense layers as convolutional ones is useful for managing images from larger sizes. Then, we pass the dense layers to convolutional ones. In this case, the network would look like figure 40.

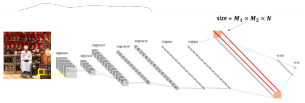


Figure 40: architecture changing dense layers for convolutional ones, with larger images

Flattening layers can be also converted to convolutional ones, so we can use regular CNNs for segmentation. Therefore, we can take advantage of the known pre-trained models This segmentation method is not accurate, but it could be useful to solve the problems without training a heavy model. Plus, the output has a very different resolution,

10.9 Heatmap Upsampling

The heatmap upsampling is an approach to solve the smaller outputs that FCNNs produce.

11 Localization

11.1 Classification and Localization tasks

This task consists of taking an image as an input with a single relevant object to be classified in a fixed set of categories. The goals are:

1. Assign the object class to the image
2. Locate the object in the image by a bounding box

It is required a training set of annotated images with labels and a bounding box around each object.

11.2 The problem

The formal problem is assigning to an input image $I \in \mathbb{R}^{R \times C \times 3}$ a label ℓ from a fixed set of categories Λ the coordinates (x,y,h,w), where (x,y) represent the top-left part of the box, while (h,w) its x,y-axis size

$$I \to (x, y, h, w, \ell)$$

11.3 Multitask Learning

As classification and localization have different natures, the network requires to have 'two heads'. It is used Mean Square Error as a loss for the bounding box and Categorical Cross-entropy for the classification, two different losses.

However, the training loss has to be a single scalar, since we compute the gradient of a scalar function. Then, we minimize the multitask loss that merges two losses using the convex combination with a hyperparameter $\alpha \in [0, 1]$.

$$\mathcal{L}(x) = \alpha S(x) + (1 - \alpha) R(x)$$

This is even able to find boxes with coordinates outside the box if, for instance, the object is cropped by the frame.

It is important to find the correct α, but taking into account that losses with different α are not comparable.
To implement this in tensorflow:

```
############
inputs = tfk.Input(shape=train_images.shape[1:])
x = mobile(inputs)
############ Add a classification head for object (binary/multiclass)
    classification
class_outputs = tfkl.Dense(2, activation='softmax', name='classifier')(x)
############ Add localization head for bounding box prediction
box_outputs = tfkl.Dense(4, activation='linear', name='localizer')(x)
############ Create the model connecting inputs to classification and
    localisation outputs (it is enough to put the output in a list)
object_localization_model = tfk.Model(inputs=inputs,
outputs=[class_outputs, box_outputs], name='object_localization_model')
```

```
############ Compile the model with categorical crossentropy and mean squared
     error losses (make sure the two losses are in the same order as the heads
    . The i-th item of the output list is assessed using the i-th item of the
    loss list
object_localization_model.compile(loss=[tfk.losses.CategoricalCrossentropy(),
     tfk.losses.MeanSquaredError()], optimizer=tfk.optimizers.Adam())
```

It is also possible to use two FCs separately, but it is always better to perform at least some fine-tuning to train localization and classification together.

11.4 Weak Supervision

In supervised learning, a model M performs inference over Y

$$M : X \to Y$$

It requires a training set $TR \subset X \times Y$.
Weak Supervision consists of obtaining a model able to solve a task in Y, but using labels that are easier to gather in a different domain K. Therefore, M after training performs inference as $M : X \to Y$ as well, but it was trained using $TR \subset X \times K$, $K \neq Y$.
Specifically, it is possible to perform localization without localization labels provided, just with image-label pairs $\{(I, \ell)\}$

12 Explainability

12.1 First filters

We are interested on how much a filter is activated and where, to explain why a network gives the results that it provides. The first layers match low-level features as edges and simple patterns that are discriminative to describe the data.

When we dive into deeper layers, often architectures use more channels, not allowing us to interpret them as images.

12.2 Maximally activating patches

We can also revise what parts of the image are activating most in specific neurons, as follows:

1. Select a neuron in a deep layer of a pre-trained CNN
2. Perform inference and store the activations for each input image
3. Select the image yielding the maximum activation
4. Show the regions (patches) corresponding to the receptive field of the neuron
5. iterate for many neurons

12.3 Input that maximally activates neuron

We can also compute which input maximizes the value of a specific activation. This can be done with a gradient ascent (instead of descent).
The image results will give us low-level patterns in shallow (or superficial) layers, but more complex patrons in deeper layers.

12.4 Input that maximally activates output

We can also look for the image that maximally activate the softmax function at the output. Given a class c, the function to optimize is

$$\hat{I} = argmax_I S_c(I) + \alpha \|I\|_2^2$$

It is necessary add a $\alpha > 0$ regularization parameter to improve the smoothness of acquired images.

12.5 Saliency maps use

Saliency maps allow us to make decisions knowing which parts of the image are important to predict a specific label. This is important as well because models can make decisions based on wrong information, despite the prediction being correct.

The head maps should be discriminative between classes and should capture fine-grained details

12.6 Class Activation Mapping (CAM)

The advantages of GAP layer extend beyond simply acting as a structural regularizer that prevents overfitting.
CNNs can retain a remarkable localization ability until the final layer. By a simple tweak it is possible to easily identify the discriminative image regions leading to a prediction. This is done by class activation mapps.
Class Activation Mapping (CAM) identifies exactly which regions of an image are being used for discrimination. It consists of:

1. A classifier trained with a GAP layer, computing a function

$$F_k = \sum_{(i,j) \in I} f_k(i,j)$$

2. A single output FC layer after GAP, with c output neurons, that computes a function

$$S_c = \sum_k w_k^c F_k$$

Note that w_k^c encodes the importance of F_k for the class c

3. A softmax activation, that computes

$$y_c = \frac{e^{S_c}}{\sum_i e^{S_i}}$$

Putting together steps 1 and 2:

$$\begin{aligned} S_c &= \sum_k w_k^c \sum_{(i,j)\in I} f_k(i,j) \\ &= \sum_k \sum_{(i,j)\in I} w_k^c f_k(i,j) \\ &= \sum_{(i,j)\in I} \sum_k w_k^c f_k(i,j) \qquad \text{(As they are finite sums)} \\ &= \sum_{(i,j)\in I} M_c(i,j) \end{aligned}$$

We defined Class Activation Mapping (CAM)

$$M_c(i,j) := \sum_k w_k^c f_k(i,j)$$

It represents the importance of the activations at (i, j) for predicting the class c.
It is often used a threshold of 20% max(CAM(i, j)) to say it detected the object (see figure 41)

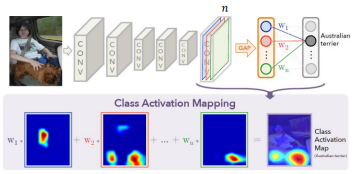


Figure 41: CAM architecture example

The CAM helps us to detect what parts of the image the neural network is using to predict a certain label

Figure 42: CAM compared with images and label predicted: example

CAM can be included in any pre-trained network as long the FC layers are replaced for GAP and a single FC layer (this could lead to a performance drop).
A GMP (Global Max Pooling) could be used as well, to promote specific discriminative features

The CAM could be implemented in Keras as follows:

```
def compute_CAM(model, img):
    ############ Expand image dimensions to fit the model input shape
    img = np.expand_dims(img, axis=0)
    ############ Predict to get the winning class
    predictions = model.predict(img, verbose=0)
    label_index = np.argmax(predictions)
    ############ Get the 1028 input weights to the softmax of the winning
    class
    ############ These are the weights of the fully connected after the GAP
    before the output
    class_weights = model.layers[-1].get_weights()[0]
    ############ These are the weights related to the winning class
    class_weights_winner = class_weights[:, label_index]
    ############ Take the MobileNetV2 until the final convolutional layer
    final_conv_layer = tfk.Model(
    model.get_layer('mobilenetv2_1.00_224').input,
    model.get_layer('mobilenetv2_1.00_224').get_layer('Conv_1').output)

    ############ Compute the convolutional outputs and squeeze the dimensions
    conv_outputs = final_conv_layer(img)
    conv_outputs = np.squeeze(conv_outputs)
    ############ Upsample the convolutional outputs
    mat_for_mult = scipy.ndimage.zoom(conv_outputs, (32, 32, 1), order=1)
    ############ Flatten the spatial dimension
    mat_for_mult = mat_for_mult.reshape((256*256, 1280))
    ############ Compute the CAM as the weighted sum of channels, using the
    weights of dense la
    yer as weights of the combination. This is the matrix variant of the
    formulas seen before, it is possible to replace this by for loops
    final_output = np.dot(mat_for_mult, class_weights_winner)
    ############ reshape the CAM
    final_output = final_output.reshape(256,256)
    return finaloutput, labelindex, predictions
```

12.7 Super Resolution Upsampling

The super-resolution upsampling (Morbidelli, Carrera, Rossi, Fragneto, & Boracchi, 2020) uses L linear augmentations A_l over the original image $x \in \mathbb{R}^{N \times M}$ to improve the results of the saliency maps:

$$x_l = A_l(x)$$

The original model uses translations and rotations, using that every heat map is much more informative than the original one only.

The modeling assumes that there exists a high-resolution heat-map h such that applying the 'downsampling operator' $D : \mathbb{R}^{N \times M} \to \mathbb{R}^{n \times m}$ we will obtain g_l, the low-resolution saliency map resulting from the model to be explained:

$$g_l \approx D \circ A_l(h)$$

Therefore, to approximate this high-resolution heat-map h, we will solve the following problem:

$$\min_h \frac{1}{2}\sum_{l=1}^{L} \|DA_l h - g_l\|_2^2 + \lambda TV_{\ell_1}(h) + \frac{\mu}{2}\|h\|_2^2$$

where TV_{ℓ_1} is the Anisotropic Total-Variation Regularization:

$$TV_{\ell_1}(h) = \sum_{i,j} |\partial_x h(i,j)| + |\partial_y h(i,j)|$$

where $\partial_x h(i,j)$ is the difference between the current and next pixel in the horizontal direction in h. $\partial_y h(i,j)$ is defined analogously.

This problem is solved through sub-gradient descent since the function is convex but non-smooth.

12.8 Other techniques

There are other CAM-based techniques to obtain the saliency maps.

Grad-CAM calculates the activation map differently. It uses the derivatives in the back-propagation to create the weights:

$$w_k^c = \frac{1}{nm}\sum_i \sum_j \frac{\partial y_c}{\partial f_k(i,j)}$$

$$g^c(i,j) = RELU\left(\sum_k w_k^c f_k(i,j)\right)$$

where:

- n,m is the spatial dimension of the layer
- y_c is the network output for the class c

Grad-CAM++ uses the same approach as Grad-CAM, but uses higher-order derivatives.

Smooth Grad-CAM++ averages multiple heat-maps corresponding to noisy versions of the same input image.

There are perturbation saliency maps as well. For example, there is RISE (Petsiuk, Das, & Saenko, 2018):

1. Samples N random masks M_i to be applied to the original image
2. Applies the mask i to pass the masked image to the model to be explained, getting y^i, the model output on the image masked by i

3. It calculates the saliency map as:

$$S(i,j) := \frac{1}{\sum_i M_i} \sum_{i=1}^{N} y_c^i \cdot M_i(i,j)$$

where y_c^i is the class c in y^i

.

13 Implementing Object Localization and Activation Maps

This section has been created based on Prof. Eugenio Lomurno's notebook

13.1 Aditional Imports

```
import cv2
import csv
import scipy
from PIL import Image
from xml.dom import minidom
import pandas as pd
import matplotlib as mpl
import matplotlib.pyplot as plt
from concurrent.futures import ThreadPoolExecutor
import seaborn as sns
from tensorflow.keras.applications.mobilenet import preprocess_input
```

13.2 Metric to measure boxes fitting

```
def spearman_rho(box_predictions, val_boxes):

    # Reshape predictions and validation boxes into 1-D tensors
    box_predictions = tf.reshape(box_predictions, [-1])
    val_boxes = tf.reshape(val_boxes, [-1])

    # Function to compute ranks of elements
    def rank(x):
        # Sort elements and obtain their indices
        sorted_indices = tf.argsort(x, direction='ASCENDING')

        # Assign ranks based on the sorted indices
        ranks = tf.argsort(sorted_indices, direction='ASCENDING') + 1

        return tf.cast(ranks, tf.float32)

    # Compute ranks for the predicted and actual boxes
    rank_pred = rank(box_predictions)
    rank_val = rank(val_boxes)

    # Calculate the mean of the ranks
    mean_rank_pred = tf.reduce_mean(rank_pred)
```

```
    mean_rank_val = tf.reduce_mean(rank_val)

    # Calculate differences from the mean ranks
    diff_pred = rank_pred - mean_rank_pred
    diff_val = rank_val - mean_rank_val

    # Compute covariance of the rank differences
    cov = tf.reduce_mean(diff_pred * diff_val)

    # Compute standard deviations of the rank differences
    std_pred = tf.sqrt(tf.reduce_mean(tf.square(diff_pred)))
    std_val = tf.sqrt(tf.reduce_mean(tf.square(diff_val)))

    # Compute Spearman's rank correlation coefficient with epsilon to avoid
    division by zero
    spearman_rho = cov / (std_pred * std_val + 1e-8)

    return spearman_rho
```

13.3 Building a Model for Multitasking

```
# Load a pretrained model, freezing its layers.
# Add a dense layer
# Define the losses as follows
class_outputs = tfkl.Dense(2, activation='softmax', name='classifier')(x)

# Add localisation head for bounding box prediction
box_outputs = tfkl.Dense(4, activation='linear', name='localizer')(x)

# Create the model connecting inputs to classification and localisation
    outputs
object_localization_model = tfk.Model(inputs=inputs, outputs=[class_outputs,
    box_outputs], name='object_localization_model')

# Compile the model with categorical crossentropy and mean squared error
    losses, using Adam optimiser
object_localization_model.compile(loss=[tfk.losses.CategoricalCrossentropy(),
     tfk.losses.MeanSquaredError()], optimizer=tfk.optimizers.Adam())
```

13.4 Loading data

```
x_train = preprocess_input(train_images)
y_train = [train_labels, train_boxes]
## The model should be trained with these data. We have to apply these
    transformations to validation data as well
```

13.5 Prediction

```
# Generate predictions on the validation images
val_predictions = object_localization_model.predict(preprocess_input(
    val_images), verbose=0)
classification_predictions = np.argmax(val_predictions[0], axis=1)
box_predictions = val_predictions[1]
```

```

# Retrieve true labels from the validation set
val_gt = np.argmax(val_labels, axis=1)

# Compute and display confusion matrix
cm = confusion_matrix(val_gt, classification_predictions)

spearman = spearman_rho(val_boxes, box_predictions)
```

13.6 Class Activation Maps

Compute CAM

```
def compute_CAM(model, img):
    # Expand image dimensions to fit the model input shape
    img = np.expand_dims(img, axis=0)

    # Predict to get the winning class
    predictions = model.predict(img, verbose=0)
    label_index = np.argmax(predictions)

    # Get the 1028 input weights to the softmax of the winning class
    class_weights = model.layers[-1].get_weights()[0]
    class_weights_winner = class_weights[:, label_index]

    # Define the final convolutional layer of the MobileNetV2 model
    final_conv_layer = tfk.Model(
        model.get_layer('mobilenetv2_1.00_224').input,
        model.get_layer('mobilenetv2_1.00_224').get_layer('Conv_1').output
    )

    # Compute the convolutional outputs and squeeze the dimensions
    conv_outputs = final_conv_layer(img)
    conv_outputs = np.squeeze(conv_outputs)

    # Upsample the convolutional outputs and compute the final output using
    the class weights
    mat_for_mult = scipy.ndimage.zoom(conv_outputs, (32, 32, 1), order=1)
    final_output = np.dot(mat_for_mult.reshape((256*256, 1280)),
    class_weights_winner).reshape(256,256)

    return final_output, label_index, predictions

values = []
for img in X_test_preprocessed:
    values.append(compute_CAM(classifier_model, img))
```

Get the CAM maps

```
# Inspect the data
num_img = 10
fig, axes = plt.subplots(2, num_img//2, figsize=(20,9))
for i in range(num_img):
    ax = axes[i%2,i%num_img//2]
    ax.imshow(values[i][0], cmap='turbo')
    ax.imshow(np.clip(X_test[i], 0, 255), alpha=0.5)
    ax.axis('off')
```

```
plt.tight_layout()
plt.show()
```

Plot rectangles based on HeatMaps

```
# Display a sample of 10 test images with heatmap-based bounding boxes and
    class labels
num_img = 10
fig, axes = plt.subplots(2, num_img // 2, figsize=(20, 9))

for i in range(num_img):

    # Extract the maximum value from the heatmap
    heatmap_max = np.max(values[i][0])

    # Define a threshold to filter heatmap values
    boundary = heatmap_max * 0.3

    # Apply the threshold to create a binary heatmap
    bbox_heatmap = values[i][0].copy()
    bbox_heatmap = np.where(bbox_heatmap <= boundary, 0, 255)

    bbox_img = X_test[i].copy()

    # Find contours of the heatmap
    cnts = cv2.findContours(bbox_heatmap.astype('uint8'),
                            cv2.RETR_EXTERNAL,
                            cv2.CHAIN_APPROX_SIMPLE)
    cnts = cnts[0] if len(cnts) == 2 else cnts[1]

    # Define an offset for the bounding box
    offset = 10
    for c in cnts:
        x, y, w, h = cv2.boundingRect(c)
        cv2.rectangle(bbox_img, (x + offset, y + offset),
                      (x + offset + w, y + h), (100, 255, 0), 3)

    # Get the predicted label and confidence score
    label = num_to_labels[values[i][1]]
    confidence = round(values[i][2][0][values[i][1]] * 100, 1)

    # Display image with bounding box and label
    ax = axes[i % 2, i % (num_img // 2)]
    ax.imshow(np.clip(bbox_img, 0, 255))
    ax.title.set_text(f'{label}: {confidence}%')
    ax.axis('off')

plt.tight_layout()
plt.show()
```

14 Object Detection

14.1 The problem

Assign to an input image $I \in \mathbb{R}^{R \times C \times 3}$ multiple labels $\{l_i\}$ a fixed set of categories Λ, each one with coordinates $\{(x, y, h, w)_i\}$ that enclose the object

14.2 Multi-label Classification

To perform a multiple classification output, we can replace softmax activation in the output layer with sigmoid activation in each of the L neurons. In this case, the ith neuron will predict the probability $p_i \in [0, 1]$ of class membership for the input, being non-exclusive between them.
This model must be trained with the binary cross-entropy loss function.

14.3 The Loss Function

To assess the network performance placing bounding boxes, it is used

$$IoU = \frac{\text{Area of intersection/overlap}}{\text{Area of union}}$$

14.4 Naive approach

To cope with this kind of problem, we can consider to use patch images that encloses different kinds of objects, so then the network will be able to detect them in larger images. Anyway this is very inefficient, since the patch sizes could be arbitrary and a lot of features can share pixels.

14.5 R-CNN

This algorithm was proposed by Girshick, Ross, et.al. in 2014, and was called R-CNN. This approach is a little faster but it is not very efficient anyway.
They designed the Region Proposal Algorithms, which are meant to identify bounding boxes that correspond to a candidate object in the image. Then, the model takes all these patches identified to use them later. The algorithm warps each image to fit it later using the next part of the model. Afterward, those objects can be feature-extracted by a CNN as patches. The classification will be performed at the network end with an SVM classifier + BB regressor. This is done because if we use a pre-trained model as a CNN, we cannot propagate the gradient to the region proposal part of the model, so a different type of training should be applied. This last part makes adjustments over the bounding boxes as well.
We must ensure to include the background class in Λ.
This model has a very high recall but low precision.

14.6 Fast R-CNN

It was proposed in 2015 by Microsoft Research. As its name says, it is way faster than R-CNN. The algorithm is the following:

1. The whole image feeds a CNN that extracts feature maps.

2. Region proposals are identified from the initial image, but they are projected into the feature maps. Regions are directly cropped from the feature maps, instead of from the image, re-using convolutional computation.

3. As a fixed size is required to feed data to fully connected layers, it extracts a fixed size $H \times W$ activation from each region proposal. Each Region of Interest (ROI) in the

feature maps is divided in a $H \times W$ grid and the maxpooling over the grid provides a fixed size input to the next step

4. The FC layers estimate both classes with softmax and BB location with the Bbox regressor. A convex combination of the two is used as a multitask loss to be optimized.

The majority of the time is used by the region extraction part.

14.7 Faster R-CNN

It was developed in 2015 by Microsoft Research team. It proposes:

1. Backbone: Feature Extractor, usually with a pre-trained network

2. Region Proposal Network: Shared convolution, plus learnable in parallel with the bounding boxes, that then concatenate.

   RPN is a F-CNN with 3x3 filter size. It operates on the same feature maps used for classification. It could be seen as an additional module that improves efficiency and focuses fast R-CNN over the most promising regions for object detection.

   One of the goals is to associate each spatial location (pixel) k anchor boxes (reference shapes). It outputs $r_b \times c_b \times k$ anchor candidates and estimates $2k$ probabilities for each one of the features maps of $r_b \times c_b$. These probabilities are called 'objectiveness score', and measure whether the box contains an object or does not. Estimation is obtained with a sigmoid activation

   The other path of the network is trained to adjust each of the k anchors to better match objects in the image. It computes four deltas (one for each coordinate) for each bounding box.

   Consequently, the combination of these two paths returns k proposals for each location in the latent space, considering a fixed threshold on the objectiveness score

   Finally, only the n_{prop} proposals are going to be propagated to the next layer, based on IoU and objectiveness score.

   Plus, there is a skip connection that goes directly from the backbone output to the detection head.

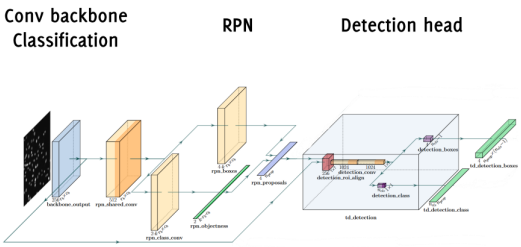


Figure 43: RPN original architecture

3. Two heads to perform classification and bounding boxes, as Fast R-CNN. We don't perform the bounding box corrections for background class

4. The training involves 4 losses, in a weighted sum:
    - RPN classify object/non-object
    - RPN regression coordinates
    - Final Classification Score
    - Final BB coordinates

The usual training procedure consists of:

1. Training RPN keeping backbone network frozen and training only RPN layers. This ignores object classes but just bounding box locations.
2. Training Fast-RCNN using proposals from RPN trained before. Fine-tune the whole Fast-RCNN (the last part) including the backbone
3. Fine-tune the RPN in cascade of the new backbone
4. Freeze backbone and RPN and fine-tune only the last layers of the Faster R-CNN

14.8 You Only Look Once (YOLO)

It was developed by researchers from University of Washington, Allen Institut for AI, and Facebook AI Research in 2016.
Previously, the location problem was tackled with region-based methods, making necessary having two steps during inference, and making it difficult to optimize as well. This architecture reframed the object detection as a single regression problem from image pixels to bounding box coordinates and class probabilities, and solve these regression problems all at once with a large CNN.
The process consists of:

1. Divide the image in a coarse grid (e.g. 7x7)
2. Each grid cell contains B anchors associated
3. For each cell and anchor is predicted:
    - The offset of the base bounding box. The deltas including the objectness score
    - The classification score of the base bounding box over the C considered categories, including background.

    The output network dimension has then

    $$grid\_heigth \times grid\_width \times B \times (5 + C)$$

    Where 5 is the 5 elements of the offset and C is the number of classes

15 Using Object Detection with Implemented Faster R-CNN

This section has been created based on Prof. Eugenio Lomurno's notebook

15.1 Aditional Imports

```
# Import additional libraries
from six.moves.urllib.request import urlopen
from six import BytesIO
from PIL import Image, ImageDraw, ImageFont
import matplotlib.pyplot as plt
import tensorflow_hub as hub
from matplotlib import cm
import seaborn as sns
import time
```

15.2 Importing Model

```
category_intex = {
    1: "person", 2: "bicycle"
} # Mapping class ID labels

# Specify the URL handle to download the model from TensorFlow Hub
model_handle = "https://tfhub.dev/tensorflow/faster_rcnn/
    inception_resnet_v2_1024x1024/1"

hub_model = hub.load(model_handle)
```

15.3 Inference

```
# Perform the object detection inference using the pre-loaded model
results = hub_model(image_np)

# Convert the results from tensors to NumPy arrays for further processing
result = {key: value.numpy() for key, value in results.items()}

# Extract class IDs, removing the batch dimension and converting to integer
    type
classes = result['detection_classes'][0].astype(int)
# Extract bounding box coordinates, removing the batch dimension
boxes = result['detection_boxes'][0]
# Extract confidence scores, removing the batch dimension
confidence_scores = result['detection_scores'][0]
```

15.4 Object Detection by confidence

```
minimum_confidence = 0.75

# Print the total number of predictions
print("Total predictions:", len(classes))

# Iterate through the first 100 predictions
for example_prediction in np.arange(len(classes)):
    # Check if the prediction's confidence score is greater than the the
    minimum confidence
    if confidence_scores[example_prediction] > minimum_confidence:
        # Print details of the high-confidence prediction
```

```
        print("Example prediction nr", example_prediction, ":")
        print("\tPredicted class:", category_index[classes[example_prediction
    ]])
        print("\tPredicted box:", boxes[example_prediction])
        print("\tModel confidence:", confidence_scores[example_prediction])
```

15.5 Plot Functions

```
# Define a unique colour for each class
colors = [tuple((color * 255).astype(int)) for color in cm.rainbow(np.
    linspace(0, 1, len(category_index)))]

# Visualise predictions with bounding boxes and related information on an
    image
def show_prediction(image, show_boxes, show_classes, show_scores,
    max_boxes_to_draw=200, figsize=(15, 15)):
    show_image = image[0] if image.shape[0] == 1 else image
    pil_image = Image.fromarray(show_image).convert('RGBA')

    image_height = show_image.shape[0]
    image_width = show_image.shape[1]
    resize_factor = [image_height, image_width, image_height, image_width]
    full_boxes = resize_factor * show_boxes

    final_boxes = full_boxes[:max_boxes_to_draw]
    final_classes = show_classes[:max_boxes_to_draw]
    final_scores = show_scores[:max_boxes_to_draw]

    font = ImageFont.load_default()
    image_draw = ImageDraw.Draw(pil_image)
    for box, cls, score in zip(final_boxes, final_classes, final_scores):
        y_0, x_0, y_1, x_1 = box
        image_draw.rectangle((x_0, y_0, x_1, y_1), outline=colors[cls], width
    =3)
        text = f"{category_index[cls]} {score:.2f}"
        text_position = (x_1 + 2, y_0)
        text_width, text_height = font.getsize(text)
        margin = np.ceil(0.05 * text_height)
        image_draw.rectangle((x_1 + 2 - margin, y_0 - margin, x_1 + 2 +
    text_width + margin, y_0 + text_height + margin), fill=colors[cls])
        text_color = "black" if sum(colors[cls][:3]) > 500 else "white"
        image_draw.text(xy=text_position, text=text, fill=text_color)

    plt.figure(figsize=figsize)
    plt.imshow(pil_image)
    plt.show()

# Define a function to count instances and calculate bounding box areas
def print_prediction_count(image, classes, boxes, label_dict=category_index):
    # Adjust the image and the boxes as in the show_prediction function
    show_image = image[0] if image.shape[0] == 1 else image
    rescaled_boxes = (boxes * [show_image.shape[0], show_image.shape[1],
                               show_image.shape[0], show_image.shape[1]]).
    astype(int)

    # Iterate through each unique class detected
```

```
    for cls in np.unique(classes):
        # Count the instances of the current class
        n_instances = sum(classes == cls)

        # Select the boxes corresponding to the current class
        class_indices = np.where(classes == cls)
        class_boxes = rescaled_boxes[class_indices]

        # Compute the areas of the boxes
        class_areas = ((class_boxes[:, 2] - class_boxes[:, 0])
                       * (class_boxes[:, 3] - class_boxes[:, 1]))
        # Compute the minimum, maximum, and average areas
        max_area = np.max(class_areas)
        min_area = np.min(class_areas)
        avg_area = np.round(np.mean(class_areas), 2)

        # Print the count and area statistics for each class
        print(f"Number of '{label_dict[cls]}' instances: {n_instances};
    minimum area: {min_area}px; maximum_area: {max_area}px; average area: {
    avg_area}px")
```

15.6 Plotting Images with object detection

```
# Create a mask for predictions that meet or exceed the minimum confidence
    threshold
over_threshold_mask = confidence_scores >= minimum_confidence

# Apply the mask to keep only the bounding boxes with sufficient confidence
final_boxes = boxes[over_threshold_mask]

# Apply the mask to keep only the class IDs with sufficient confidence
final_classes = classes[over_threshold_mask]

# Apply the mask to keep only the confidence scores that meet the threshold
final_scores = confidence_scores[over_threshold_mask]

# Call the function to print prediction counts and area statistics
print_prediction_count(image_np, final_classes, final_boxes)

# Call the function to show predictions on the image with the final set of
    filtered bounding boxes,
# class labels, and confidence scores
show_prediction(image_np, final_boxes, final_classes, final_scores)
```

16 Instance Segmentation

16.1 The problem

In contrast to Object Detection, Instance Segmentation does not try to fit rectangular boxes, but classify each of the pixels of the image, giving to a group of them a specific label. Particularly, for an input image I, assign:

- Multiple labels $\{l_i\} \in \Lambda$, assigning an instance of that category as well

- The coordinates $\{(x, y, h, w)_i\}$ of the bounding box enclosing each object
- The set of pixels S in each bounding box corresponding to that label

$$I \rightarrow \{(x, y, h, w, l, S)_1, \ldots, (x, y, h, w, l, S)_N\}$$

It can identify and mark two overlapped objects as different instances, even when they are part of the same class

16.2 Mask R-CNN

It was created in 2017 by Facebook AI Research (FAIR). It combined object detection with semantic segmentation, adding this last one at the end of the Faster R-CNN architecture

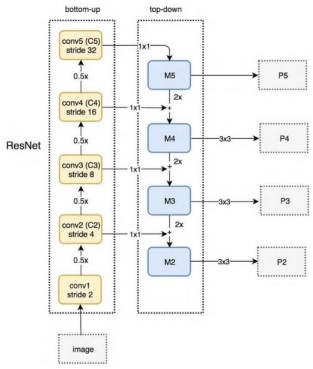


Figure 44: Masked R-CNN architecture

16.3 Feature Pyramid Network

The goal of this network is to leverage the pyramidal shape of a ConvNet's feature hierarchy while creating a feature pyramid that has strong semantics at all scales. It startes to do predictions independently on each level of the architecture.
The architecture is compounded of two parts (see figure 45):

1. **Bottom-up:** It performs convolutions that reduce the spatial dimension, originally, to the half of the size.
2. **Top-down:** It derives higher resolution by upsampling, originally increasing size with a factor of 2.
3. **Connection:** It does skip connections between each matched part of the bottom-up and top-down stage, doing 1x1 convolutions to reduce channel dimension, and adding it with the result of the upsampling
4. **Prediction:** it attaches a predictor head in each level of the feature pyramid, on the top-down part, doing unnecessarily different multi-scale anchors.

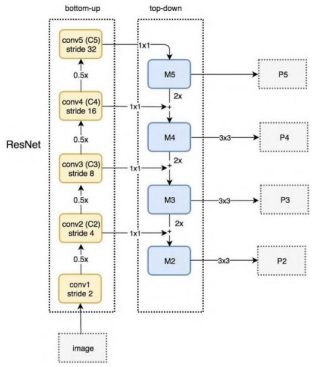


Figure 45: FPN

17 Implementing Semantic Segmentation with UNet

This section has been created based on Prof. Eugenio Lomurno's notebook

17.1 Saving resources

```
def apply_category_mapping(label):
    """
    Apply category mapping to labels.
    """
    category_map = {
        # Here we remap the number classes so we can do our model less complex, using no so much computation.
    }
    keys_tensor = tf.constant(list(category_map.keys()), dtype=tf.int32)
    vals_tensor = tf.constant(list(category_map.values()), dtype=tf.int32)
    table = tf.lookup.StaticHashTable(
        tf.lookup.KeyValueTensorInitializer(keys_tensor, vals_tensor),
        default_value=0
    )
    return table.lookup(label)

def load_single_image(image_path, label_path, input_size=(64, 128)):
    """
    Load a single image-label pair with the correct shape.
    """
    # Read and preprocess the image
    image = tf.io.read_file(image_path)
    image = tf.io.decode_png(image, channels=3)  # Ensure 3 channels
    image = tf.image.resize(image, input_size)   # Resize to fixed size
    image = tf.cast(image, tf.float32) / 255.0

    # Read and preprocess the label
    label = tf.io.read_file(label_path)
    label = tf.io.decode_png(label, channels=1)  # Ensure single channel
    label = tf.image.resize(label, input_size, method='bilinear')  # Resize to fixed size
    label = tf.cast(label, tf.int32)

```

```
    return image, label
```

17.2 Model building

```
def unet_block(input_tensor, filters, kernel_size=3, activation='relu', stack=2, name=''):
    # Initialise the input tensor
    x = input_tensor

    # Apply a sequence of Conv2D, Batch Normalisation, and Activation layers for the specified number of stacks
    for i in range(stack):
        x = tfkl.Conv2D(filters, kernel_size=kernel_size, padding='same', name=name + 'conv' + str(i + 1))(x)
        x = tfkl.BatchNormalization(name=name + 'bn' + str(i + 1))(x)
        x = tfkl.Activation(activation, name=name + 'activation' + str(i + 1))(x)

    # Return the transformed tensor
    return x

def get_unet_model(input_shape=(64, 128, 3), num_classes=NUM_CLASSES, seed=seed):
    tf.random.set_seed(seed)
    input_layer = tfkl.Input(shape=input_shape, name='input_layer')

    # Downsampling path
    down_block_1 = unet_block(input_layer, 32, name='down_block1_')
    d1 = tfkl.MaxPooling2D()(down_block_1)

    down_block_2 = unet_block(d1, 64, name='down_block2_')
    d2 = tfkl.MaxPooling2D()(down_block_2)

    # Bottleneck
    bottleneck = unet_block(d2, 128, name='bottleneck')

    # Upsampling path
    # UpSampling2D() is not the unique possible layer, but it is the opposite operation to MaxPooling2D()
    u1 = tfkl.UpSampling2D()(bottleneck)
    u1 = tfkl.Concatenate()([u1, down_block_2]) # We can use an addition as well, it is a trade-off with information and resources.
    u1 = unet_block(u1, 64, name='up_block1_')

    u2 = tfkl.UpSampling2D()(u1)
    u2 = tfkl.Concatenate()([u2, down_block_1])
    u2 = unet_block(u2, 32, name='up_block2_')

    # Output Layer
    output_layer = tfkl.Conv2D(num_classes, kernel_size=1, padding='same', activation="softmax", name='output_layer')(u2)

    model = tf.keras.Model(inputs=input_layer, outputs=output_layer, name='UNet')
    return model

```

```
model = get_unet_model()
```

17.3 Defining Personalized parameters

```
# Define custom Mean Intersection Over Union metric
class MeanIntersectionOverUnion(tf.keras.metrics.MeanIoU):
    def __init__(self, num_classes, labels_to_exclude=None, name="mean_iou", dtype=None):
        super(MeanIntersectionOverUnion, self).__init__(num_classes=num_classes, name=name, dtype=dtype)
        if labels_to_exclude is None:
            labels_to_exclude = [0]  # Default to excluding label 0
        self.labels_to_exclude = labels_to_exclude

    def update_state(self, y_true, y_pred, sample_weight=None):
        # Convert predictions to class labels
        y_pred = tf.math.argmax(y_pred, axis=-1)

        # Flatten the tensors
        y_true = tf.reshape(y_true, [-1])
        y_pred = tf.reshape(y_pred, [-1])

        # Apply mask to exclude specified labels
        for label in self.labels_to_exclude:
            mask = tf.not_equal(y_true, label)
            y_true = tf.boolean_mask(y_true, mask)
            y_pred = tf.boolean_mask(y_pred, mask)

        # Update the state
        return super().update_state(y_true, y_pred, sample_weight)

# Visualization callback
class VizCallback(tf.keras.callbacks.Callback):
    def __init__(self, image_path, label_path, frequency=5):
        super().__init__()
        self.image_path = image_path
        self.label_path = label_path
        self.frequency = frequency

    def on_epoch_end(self, epoch, logs=None):
        if epoch % self.frequency == 0:  # Visualize only every "frequency" epochs
            image, label = load_single_image(self.image_path, self.label_path)
            label = apply_category_mapping(label)
            image = tf.expand_dims(image, 0)
            pred = self.model.predict(image, verbose=0)
            y_pred = tf.math.argmax(pred, axis=-1)
            y_pred = y_pred.numpy()

            # Create colormap
            num_classes = NUM_CLASSES
            colormap = create_segmentation_colormap(num_classes)

            plt.figure(figsize=(16, 4))

```

```
            # Input image
            plt.subplot(1, 3, 1)
            plt.imshow(image[0])
            plt.title("Input Image")
            plt.axis('off')

            # Ground truth
            plt.subplot(1, 3, 2)
            colored_label = apply_colormap(label.numpy(), colormap)
            plt.imshow(colored_label)
            plt.title("Ground Truth Mask")
            plt.axis('off')

            # Prediction
            plt.subplot(1, 3, 3)
            colored_pred = apply_colormap(y_pred[0], colormap)
            plt.imshow(colored_pred)
            plt.title("Predicted Mask")
            plt.axis('off')

            plt.tight_layout()
            plt.show()
            plt.close()
```

17.4 Training

```
# Compile the model
print("Compiling model...")
model.compile(
    loss=tf.keras.losses.SparseCategoricalCrossentropy(),
    optimizer=tf.keras.optimizers.AdamW(LEARNING_RATE),
    metrics=["accuracy", MeanIntersectionOverUnion(num_classes=NUM_CLASSES,
    labels_to_exclude=[0])]
)
print("Model compiled!")

# Setup callbacks
early_stopping = tf.keras.callbacks.EarlyStopping(
    monitor='val_accuracy',
    mode='max',
    patience=PATIENCE,
    restore_best_weights=True
)

viz_callback = VizCallback(val_img[0], val_lbl[0])

# Train the model
history = model.fit(
    train_dataset,
    epochs=EPOCHS,
    validation_data=val_dataset,
    callbacks=[early_stopping, viz_callback],
    verbose=1
).history

# Calculate and print the final validation accuracy
```

```
final_val_meanIoU = round(max(history['val_mean_iou'])* 100, 2)
print(f'Final validation Mean Intersection Over Union: {final_val_meanIoU}%')
```

17.5 Evaluate

```
test_loss, test_accuracy, test_mean_iou = model.evaluate(test_dataset,
    verbose=0, batch_size=10)
print(f'Test Accuracy: {round(test_accuracy, 4)}')
print(f'Test Mean Intersection over Union: {round(test_mean_iou, 4)}')
```

18 Metric Learning

18.1 The problem

It consists of identifying an element with a certain class

18.2 Approaches

- We can use a trained CNN to perform classification over the classes, we only need a few images per class. In different conditions, we can satisfy this with data augmentation. The problem with this approach is that when we want to add a new class, we have to train the network from 0
- We can just choose the image that has the minimum distance with the input. The problem with this is that image distances are not a good representation of the actual meaning of the image

18.3 More Accurate Approach

A more accurate approach could be perform feature extraction of the images, providing meaningful description of the image in terms of patterns, and then compute the distance. The method consists of pass through the same network the images in class target and the input, those two networks are called siamese networks.
The contrastive loss function can be used in this problem, and it is defined as follows

$$W = argmin_\omega \sum_{i,j} \mathcal{L}_\omega(I_i, I_j, y_{i,j})$$

where:

- $\mathcal{L}_\omega(I_i, I_j, y_{i,j}) = \frac{1-y_{i,j}}{2}\|f_\omega(I_i) - f_\omega(I_j)\|_2 + \frac{y_{i,j}}{2} max(0, m - \|f_\omega(I_i) - f_\omega(I_j)\|_2)$
- $y_{i,j} \in \{0, 1\}$ is the label associated with the input pair (I_i, I_j), being 0 if they refer to the same person and 1 otherwise
- m is a hyperparameter indicating the margin we want (like in Hinge Loss)
- $\|f_\omega(I_i) - f_\omega(I_j)\|_2$ is the distance in the latent space

On the other hand, triplet loss could be used as well, where we try to minimize the distance from the positive samples and maximize the distance from the negative ones:

$$\mathcal{L}_w(I, P, N) = max(0, m + (\|f_\omega(I) - f_\omega(P)\|_2 - \|f_\omega(I) - f_\omega(N)\|_2))$$

where:

- P is a positive input, referring to the same person
- N is a negative input, referring to a different person
- m is the margin desidered

The process to verify a new image is the following

1. Feed the image I to the train network, in other words, compute $f_W(I)$
2. Identify the person having an average minimum distance from templates.

$$\hat{i} = argmin_u \frac{\sum_{T_{u,j}} \|f_W(I) - f_W(T_{u,j})\|_2}{\#\{T_u\}}$$

3. Assess whether

$$\frac{\sum_{T_{u,j}} \|f_W(I) - f_W(T_{u,j})\|_2}{\#\{T_u\}} < \gamma$$

is sufficiently small, otherwise no identification is done

19 Auto Enconders

19.1 AutoEncoders with Dense Layers

Autoencoders are used for data reconstruction (unsupervised learning), in other words, learning the identity mapping. It involves reducing and increasing the dimensionality of the layers and representing images in a way smaller space. It has an encoder $\mathcal{E}$ and a decoder $\mathcal{D}$.
In this case, the loss used is:

$$\ell(s) = \sum_{s \in S} \|s - \mathcal{D}(\mathcal{E}(s))\|_2$$

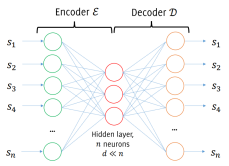


Figure 46: Autoencoder using MLP example

features $z = \mathcal{E}(s)$ are typically called latent representations. While more dimensions this representation has, the more expressive could be the space and therefore may be more faithful to the original input

19.2 Convolutional Autoencoders

We can use convolutional layers to build autoencoders as well. They are often used for images and have as well an encoder and decoder part.
When we store the latent representations, often are done in vectors. It is important to apply flattening and Reshape correctly so the result is the expected

19.3 Autoencoders for classifier initialization

Autoencoders can be used to initialize a classifier when the training set includes a lot of data without labels. In this case, we follow the next process:

1. Train the autoencoder unsupervisedly with the unlabeled data.
2. Get rid of the decoder and keep the encoder weights
3. Concatenate a classifier (FC layer) from the latent representation
4. Fine-tune the autoencoder using the few supervised samples provided. If L is large enough, the encoder can be also fine-tuned.

Using autoencoders provides a good initialization and reduces the risk of overfitting.

19.4 Autoencoders as generative models

Another use for autoencoders is to generate new data. The model could be trained, then discard the encoder, and finally, sample random vectors $z \sim \phi_z$ and feed with this the decoder.
This approach does not work well since we don't know the distribution of the latent representation (or it is difficult to estimate). Further ways to generate new data are introduced in section 21

20 Autoencoders implementation

This section has been created based on Prof. Eugenio Lomurno's notebook

20.1 Extra Imports

```
from sklearn.manifold import TSNE
import plotly.graph_objects as go
import plotly.express as px
```

20.2 Formating data

```
# Load the data
(X_train_val, y_train_val), (X_test, y_test) = tfk.datasets.mnist.load_data()

# Add channel dimension before resizing (smart_resize expects shape [batch,
    height, width, channels])
X_train_val = X_train_val[..., np.newaxis]
```

```
X_test = X_test[..., np.newaxis]

# Resize to 32x32
X_train_val = tfk.preprocessing.image.smart_resize(X_train_val, (32, 32))
X_test = tfk.preprocessing.image.smart_resize(X_test, (32, 32))
```

20.3 Build model

```
def get_encoder(enc_input_shape=input_shape, enc_output_shape=latent_dim,
    seed=seed):
    # Set the random seed for reproducibility
    tf.random.set_seed(seed)

    # Define the input layer with the specified shape
    input_layer = tfkl.Input(shape=enc_input_shape, name='input_layer')
    # We will use convolutions increasing canals but, decreasing spatial
    dimension with kernel size 3 and stride 2
    # First convolution block
    x = tfkl.Conv2D(32, 3, 2, padding='same')(input_layer)
    x = tfkl.BatchNormalization()(x)
    x = tfkl.LeakyReLU()(x)

    # Second convolution block
    x = tfkl.Conv2D(64, 3, 2, padding='same')(x)
    x = tfkl.BatchNormalization()(x)
    x = tfkl.LeakyReLU()(x)

    # Third convolution block
    x = tfkl.Conv2D(128, 3, 2, padding='same')(x)
    x = tfkl.BatchNormalization()(x)
    x = tfkl.LeakyReLU()(x)

    # Flatten the output of the last convolution layer
    x = tfkl.Flatten()(x)
    # Define the output layer with the specified output shape
    output_layer = tfkl.Dense(enc_output_shape, name='output_layer')(x)

    # Create the model linking input and output
    model = tfk.Model(inputs=input_layer, outputs=output_layer, name='encoder
    ')

    return model

def get_decoder(dec_input_shape=(latent_dim,), dec_output_shape=input_shape,
    seed=seed):
    # Set the random seed for reproducibility
    tf.random.set_seed(seed)

    # Define the input layer with the specified shape
    input_layer = tfkl.Input(shape=dec_input_shape, name='input_layer')

    # Start the decoder network with a dense layer
    # and reshape into the desired initial convolutional shape
    x = tfkl.Dense(4*4*128)(input_layer)
    x = tfkl.BatchNormalization()(x)
    x = tfkl.LeakyReLU()(x)
```

```
    x = tfkl.Reshape((4, 4, 128))(x)

    # First upsampling and convolution block
    x = tfkl.UpSampling2D(interpolation='bilinear')(x)
    x = tfkl.Conv2D(128, 3, padding='same')(x)
    x = tfkl.BatchNormalization()(x)
    x = tfkl.LeakyReLU()(x)

    # Second upsampling and convolution block
    x = tfkl.UpSampling2D(interpolation='bilinear')(x)
    x = tfkl.Conv2D(64, 3, padding='same')(x)
    x = tfkl.BatchNormalization()(x)
    x = tfkl.LeakyReLU()(x)

    # Third upsampling and convolution block
    x = tfkl.UpSampling2D(interpolation='bilinear')(x)
    x = tfkl.Conv2D(32, 3, padding='same')(x)
    x = tfkl.BatchNormalization()(x)
    x = tfkl.LeakyReLU()(x)

    # Apply a final convolution with the number of channels equal to the
    original image depth
    x = tfkl.Conv2D(dec_output_shape[-1], 3, padding='same')(x)

    # Sigmoid activation to ensure output values between 0 and 1
    output_layer = tfkl.Activation('sigmoid')(x)

    # Connect input and output through the Model class
    model = tfk.Model(inputs=input_layer, outputs=output_layer, name='decoder
    ')

    return model

def get_autoencoder(ae_input_shape=input_shape, ae_output_shape=input_shape):
    # Set the random seed to ensure reproducibility
    tf.random.set_seed(seed)

    # Initialize the encoder and decoder models
    encoder = get_encoder()
    decoder = get_decoder()

    # Define the input layer
    input_layer = tfkl.Input(shape=ae_input_shape)

    # Pass input through the encoder to get the compressed representation
    z = encoder(input_layer)

    # Pass the representation through the decoder to reconstruct the input
    output_layer = decoder(z)

    # Create the autoencoder model
    model = tfk.Model(inputs=input_layer, outputs=output_layer, name='
    autoencoder')
    return model

######################

```

```
autoencoder = get_autoencoder()
optimizer = tf.optimizers.Adam(learning_rate)

# Compile the autoencoder with Adam optimizer and mean squared error loss
autoencoder.compile(optimizer=optimizer, loss=tfk.losses.MeanSquaredError())

# Autoencoder.fit()
```

20.4 Visualize

```
def plot_label_clusters(encoder, data, labels, samples=10000):
    # Generate latent space representations
    z_mean = encoder.predict(data[:samples], verbose=0)

    # Apply TSNE if dimensions > 2
    if z_mean.shape[-1] != 2:
        tsne = TSNE(n_components=2, random_state=42)
        z_mean = tsne.fit_transform(z_mean)

    # Create figure object explicitly
    fig = go.Figure()

    # Add traces for each label
    unique_labels = sorted(set(labels[:samples]))
    colors = px.colors.qualitative.D3[:len(unique_labels)]

    for label, color in zip(unique_labels, colors):
        mask = labels[:samples] == label
        fig.add_trace(go.Scatter(
            x=z_mean[mask, 0],
            y=z_mean[mask, 1],
            mode='markers',
            name=str(label),
            marker=dict(
                size=5,
                color=color,
                line=dict(width=0)
            ),
            showlegend=True
        ))

    # Configure layout with explicit positioning
    fig.update_layout(
        # Basic setup
        showlegend=True,
        template='plotly_white',
        width=950,
        height=800,

        # Title configuration
        title=dict(
            text='Latent Space Visualization',
            font=dict(size=22),
            pad=dict(t=20),
            yref='container',
            y=0.95
```

```
        ),

        # Axes configuration
        xaxis=dict(
            title='First Component',
            showgrid=True,
            gridwidth=1,
            gridcolor='lightgray',
            zeroline=False,
            title_font=dict(size=16),
            tickfont=dict(size=14),
            title_standoff=15
        ),
        yaxis=dict(
            title='Second Component',
            showgrid=True,
            gridwidth=1,
            gridcolor='lightgray',
            zeroline=False,
            title_font=dict(size=16),
            tickfont=dict(size=14),
            title_standoff=15
        ),

        # Legend configuration
        legend=dict(
            title=dict(text='Labels'),
            yanchor='top',
            y=0.99,
            xanchor='right',
            x=0.99,
            bgcolor='rgba(255, 255, 255, 0.8)',
            bordercolor='lightgray',
            borderwidth=1,
            itemsizing='constant',
            font=dict(size=14)
        ),

        # Margins
        margin=dict(l=80, r=80, t=100, b=80)
    )

    return fig

# Project training data into latent space
plot_label_clusters(encoder, X_train, y_train)

def plot_latent_space_optimized(decoder, x_lim, y_lim, n_per_dim=20,
    digit_size=32):
    # Generate a grid of values within provided x and y limits
    grid_x = np.linspace(x_lim[0], x_lim[1], n_per_dim)
    grid_y = np.linspace(y_lim[0], y_lim[1], n_per_dim)[::-1]
    xv, yv = np.meshgrid(grid_x, grid_y)

    # Create latent vectors for each grid point
    latent_dim = decoder.input_shape[-1]
```

```
    z_samples = np.stack([np.linspace(x, y, latent_dim) for x, y in zip(xv.
    flatten(), yv.flatten())])

    # Decode the latent vectors into images
    x_decoded = decoder.predict(z_samples, verbose=0)
    digits = x_decoded.reshape(n_per_dim, n_per_dim, digit_size, digit_size)

    # Combine the individual images into a single large image
    figure = np.block([[digits[i, j] for j in range(n_per_dim)] for i in
    range(n_per_dim)])

    # Plot the large image using Plotly
    fig = go.Figure(data=go.Heatmap(
        z=figure,
        x=np.round(grid_x, 1),
        y=np.round(grid_y, 1),
        colorscale="Gray",
        showscale=False
    ))

    fig.update_layout(
        title='Latent Space - Uniform Samples',
        xaxis_title='$z_0$',
        yaxis_title='$z_1$',
        title_font_size=20,
        font=dict(
            size=16,
            color="black"
        ),
        title_x=0.5,
        width=950,
        height=950
    )

    fig.show()

# Sample data from latent space
plot_latent_space_optimized(decoder, [-5, 5], [-5, 5])
```

If we do a latent space of 2, we can see a lot of overlapping between classes, making the generation imprecise. Instead, if we do a bigger latent space, of 64, and then we project the results in 2D, we can see the classes way more separated

21 Generative Models for Images

21.1 Introduction

The goal is given a training set of images $TR = \{x_i\}$, generate other images that are similar to those in TR.
The problem is that images live in a very difficult-to-describe manifold, subspace, in a huge dimensional space.
Generative models can be used for data augmentation, simulation, and planning.

21.2 Generative Adversarial Networks (GAN) Approach

GANs do not look for an explicit density model ϕ_S describing the manifold of natural images. It just finds out a model able to generate samples that look like training samples $S \subset \mathbb{R}^n$

Instead of sampling from ϕ_S, it just samples a seed from a known distribution ϕ_Z (Often Gaussian Distribution). This is defined as a priori and also referred to as noise. GANs feed this seed with a learned transformation that generates realistic samples, as if they were drawn from ϕ_S.

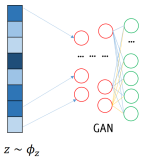


Figure 47: GAN basic architecture

21.3 $\mathcal{G}$ Loss function

Since it is difficult to assess whether an image is real or not, GAN resorts to another neural network to define it.

GAN trains a pair of neural networks addressing two different tasks that compete in a sort of two-player game (adversarial).

The first model is called generator $\mathcal{G}$, which produces realistic samples. The second model is called discriminator $\mathcal{D}$, which takes as input an image and assesses whether it is real or generated by $\mathcal{G}$. We should train both, but only $\mathcal{G}$ is kept at the end.

Therefore, $\mathcal{G}$ is trained to generate images that can fool $\mathcal{D}$, namely can be classified as real by $\mathcal{D}$. At the end of the training, we hope $\mathcal{G}$ to succeed in fooling $\mathcal{D}$ consistently

21.4 $\mathcal{D}$ Loss function

The networks are functions:

$$\mathcal{D} = \mathcal{D}(s, \theta_d)$$
$$\mathcal{G} = \mathcal{G}(z, \theta_g)$$

Where:

- θ_g and θ_d are network parameters
- $s \in \mathbb{R}^n$ is an input
- $z \in \mathbb{R}^d$ is some random noise to be fed to the generator

The networks give those outputs:

- $\mathcal{D}(\cdot, \theta_d) : \mathbb{R}^n \to [0, 1]$ gives the posterior for the input to be a true image
- $\mathcal{G}(\cdot, \theta_g) : \mathbb{R}^d \to \mathbb{R}^n$ gives the generated image

A good discriminator is such:

- $\mathcal{D}(s, \theta_s)$ is maximum when s is part of the training set.
- $1 - \mathcal{D}(s, \theta_d)$ is maximum, 1, when s was generated from $\mathcal{G}$. In other words: $1 - \mathcal{D}(\mathcal{G}(z, \theta_g), \theta_d)$ is 1 when $z \sim \phi_z$

For this reason, training $\mathcal{D}$ consists in maximizing the binary cross-entropy

$$\max_{\theta_d}(E_{s\sim\phi_S}[\log \mathcal{D}(s, \theta_d)] + E_{z\sim\phi_Z}[\log(1 - \mathcal{D}(\mathcal{G}(z, \theta_g), \theta_d))])$$

A good generator $\mathcal{G}$ makes $\mathcal{D}$ to fail, thus minimizes the above

$$\min_{\theta_g}\max_{\theta_d}(E_{s\sim\phi_S}[\log \mathcal{D}(s, \theta_d)] + E_{z\sim\phi_Z}[\log(1 - \mathcal{D}(\mathcal{G}(z, \theta_g), \theta_d))])$$

21.5 Training Process

An iterative numerical approach is needed to solve the optimization problem introduced in section 21.4.
One approach could be alternate between:

- k-steps of stochastic gradient ascent w.r.t. θ_d, keeping θ_g fixed and solve

$$\max_{\theta_d}(E_{s\sim\phi_S}[\log \mathcal{D}(s, \theta_d)] + E_{z\sim\phi_Z}[\log(1 - \mathcal{D}(\mathcal{G}(z, \theta_g), \theta_d))])$$

- 1-step of stochastic gradient descent w.r.t. θ_g, keeping θ_d fixed and solve

$$\min_{\theta_g}(E_{s\sim\phi_S}[\log \mathcal{D}(s, \theta_d)] + E_{z\sim\phi_Z}[\log(1 - \mathcal{D}(\mathcal{G}(z, \theta_g), \theta_d))])$$

  and since the first term does not depend on θ_g, this is the same as minimizing

$$\min_{(} E_{z\sim\phi_Z}[\log(1 - \mathcal{D}(\mathcal{G}(z, \theta_g), \theta_d))])$$

During early stages, $\mathcal{G}$ is poor, then $\mathcal{D}$ can reject samples with high confidence, because they are clearly different from training data. In this case, minimizing $\log(1 - \mathcal{D}(\mathcal{G}(z)))$ is flat (close to 0), thus has a very low gradient.
Therefore, for the optimizing process, we maximize $\log(\mathcal{D}(\mathcal{G}(z)))$, which is equivalent.
Therefore, the proper algorithm used is the algorithm 2

Algorithm 2 GAN training

Require: $s \sim \phi_S,\ z \sim \phi_Z$

Ensure: $\theta_d,\ \theta_g$

initialize θ_d and θ_g

for i=1,..,n_epochs **do**

for k times **do**

Draw a minibatch $\{z_1, \ldots, z_m\}$ of noise

Sample a minibatch of images $\{s_1, \ldots, s_m\}$

Update θ_d by stochastic gradient ascent:

$$\nabla\theta_d \left[\sum_i \log \mathcal{D}(s_i, \theta_d) + \log(1 - \mathcal{D}(\mathcal{G}(z_i, \theta_g), \theta_d)) \right]$$

end for

Draw a minibatch $\{z_1, \ldots, z_m\}$ of noise realization ▷ Gradient Descent steps for θ_g

Update $\mathcal{G}$ by stochastic gradient ascent

$$\nabla\theta_g \left[\sum_i \log(\mathcal{D}(\mathcal{G}(z_i, \theta_g), \theta_d)) \right]$$

end for

21.6 Some remarks:

- The training is rather unstable, we need to carefully synchronize the two alternating steps (e.g.: Wasserstein GAN).
- Selecting two reasonable noise realizations z_1 and z_2, we could interpolate among the two to generate intermediate values that are likely to be reasonable as well.
- When we have different elements for each class, we can operate between the vectors to obtain an average between all the instances. We can even operate between different classes to obtain new ones. Example: smiling woman - neutral woman + neural man = smiling man.
- We can take a mathematical ball around the image and it is still the same class.

21.7 Conditional GANs

Conditional GANs (or cGAN) consists of the same GAN approach, but conditioning the inputs with certain external user-given information (For example, the desired image label). In the case we conditionate over the label, the loss changes to:

$$\min_G \max_D L(G, D) = \mathbb{E}_{x \sim p_r(x|y)} [\log(D(x|y))] + \mathbb{E}_{x \sim p_g(x|y)} [\log(1 - D(x|y))]$$

where:

- $p_r(x|y)$ is the conditional probability of real data samples

- $p_g(x|y)$ is the conditional probability of generated samples (usually in part a random noise and other part the user-defined condition)

22 GANs implementation

This section has been created based on Prof. Eugenio Lomurno's notebook

22.1 Import additional libraries

```
from PIL import Image
import glob
import shutil
```

22.2 Building network

```
def get_discriminator(input_shape, seed=seed):
    # Set random seed for reproducibility
    tf.random.set_seed(seed)

    # Define input layer
    input_layer = tfkl.Input(shape=input_shape, name='input_layer')

    # First convolutional block
    x = tfkl.Conv2D(32, 4, padding='same', strides=2, name='conv1')(
    input_layer)
    x = tfkl.LayerNormalization(name='ln1')(x)
    x = tfkl.LeakyReLU(alpha=0.2, name='activation1')(x)

    # Second convolutional block
    x = tfkl.Conv2D(64, 4, padding='same', strides=2, name='conv2')(x)
    x = tfkl.LayerNormalization(name='ln2')(x)
    x = tfkl.LeakyReLU(alpha=0.2, name='activation2')(x)

    # Third convolutional block
    x = tfkl.Conv2D(128, 4, padding='same', strides=2, name='conv3')(x)
    x = tfkl.LayerNormalization(name='ln3')(x)
    x = tfkl.LeakyReLU(alpha=0.2, name='activation3')(x)

    # Global average pooling and dense layers for classification
    x = tfkl.GlobalAveragePooling2D(name='gap')(x)
    x = tfkl.Dense(256, name='dense1')(x)
    x = tfkl.LayerNormalization(name='ln4')(x)
    x = tfkl.LeakyReLU(alpha=0.2, name='activation4')(x)
    output_layer = tfkl.Dense(1, name='dense_out')(x)

    # Return the discriminator model
    return tf.keras.Model(inputs=input_layer, outputs=output_layer, name='
    discriminator')

def get_generator(input_shape, seed=seed):
    # Set random seed for reproducibility
    tf.random.set_seed(seed)

```

```
    # Define input layer
    input_layer = tfkl.Input(shape=(input_shape,), name='Input')

    # Dense layer to expand input to a 4x4x64 feature map
    x = tfkl.Dense(4 * 4 * 64, use_bias=False, name='dense0')(input_layer)
    x = tfkl.LayerNormalization(name='ln0')(x)
    x = tfkl.LeakyReLU(alpha=0.2, name='activation0')(x)
    x = tfkl.Reshape((4, 4, 64))(x)

    # First upsampling block
    x = tfkl.UpSampling2D(name='upsampling1')(x)
    x = tfkl.Conv2D(64, 3, padding='same', use_bias=False, name='conv1')(x)
    x = tfkl.LayerNormalization(name='bn1')(x)
    x = tfkl.LeakyReLU(alpha=0.2, name='activation1')(x)

    # Second upsampling block
    x = tfkl.UpSampling2D(name='upsampling2')(x)
    x = tfkl.Conv2D(128, 3, padding='same', use_bias=False, name='conv2')(x)
    x = tfkl.LayerNormalization(name='bn2')(x)
    x = tfkl.LeakyReLU(alpha=0.2, name='activation2')(x)

    # Third upsampling block
    x = tfkl.UpSampling2D(name='upsampling3')(x)
    x = tfkl.Conv2D(256, 3, padding='same', use_bias=False, name='conv3')(x)
    x = tfkl.LayerNormalization(name='bn3')(x)
    x = tfkl.LeakyReLU(alpha=0.2, name='activation3')(x)

    # Output layer with a single channel and tanh activation
    x = tfkl.Conv2D(1, 3, padding='same', use_bias=False, name='conv_out')(x)
    output_layer = tfkl.Activation('tanh', name='activation_out')(x) # So
    outputs are in [-1, 1]

    # Return the generator model
    model = tfk.Model(inputs=input_layer, outputs=output_layer, name='
    generator')
    return model
```

22.3 GAN class

```
class GAN(tfk.Model):
    # Initialise the GAN with a discriminator, generator, latent dimension,
    and discriminator update frequency
    def __init__(self, discriminator, generator, latent_dim,
    n_discriminator_updates=3):

        super(GAN, self).__init__()
        self.discriminator = discriminator
        self.generator = generator
        self.latent_dim = latent_dim
        # This parameter gives how many times the discriminator updates for
    each time the generator updates
        self.n_discriminator_updates = n_discriminator_updates

        # Initialise loss trackers for discriminator and generator
        self.d_loss_tracker = tfk.metrics.Mean(name="d_loss")
        self.g_loss_tracker = tfk.metrics.Mean(name="g_loss")
```

```
15
16      # Compile the GAN with optimisers and an optional loss function
17      def compile(self, d_optimizer, g_optimizer, loss_fn=None):
18          super(GAN, self).compile()
19          self.d_optimizer = d_optimizer
20          self.g_optimizer = g_optimizer
21          self.loss_fn = loss_fn or tfk.losses.BinaryCrossentropy(from_logits=
    True)
22
23      # Define GAN metrics
24      @property
25      def metrics(self):
26          return [self.d_loss_tracker, self.g_loss_tracker]
27
28      # Compute generator loss
29      def _generator_loss(self, fake_output):
30          return self.loss_fn(tf.ones_like(fake_output), fake_output)
31
32      # Compute discriminator loss
33      def _discriminator_loss(self, real_output, fake_output):
34          real_loss = self.loss_fn(tf.ones_like(real_output), real_output)
35          fake_loss = self.loss_fn(tf.zeros_like(fake_output), fake_output)
36          return real_loss + fake_loss
37
38      # Define the training step
39      @tf.function
40      def train_step(self, real_images):
41          batch_size = tf.shape(real_images)[0]
42
43          # Train discriminator multiple times
44          d_loss = 0
45          for _ in range(self.n_discriminator_updates):
46              random_latent_vectors = tf.random.normal(shape=(batch_size, self.
    latent_dim))
47
48              with tf.GradientTape() as tape:
49                  generated_images = self.generator(random_latent_vectors,
    training=True)
50                  real_output = self.discriminator(real_images, training=True)
51                  fake_output = self.discriminator(generated_images, training=
    True)
52                  current_d_loss = self._discriminator_loss(real_output,
    fake_output)
53
54              # Update discriminator weights
55              grads = tape.gradient(current_d_loss, self.discriminator.
    trainable_weights)
56              self.d_optimizer.apply_gradients(zip(grads, self.discriminator.
    trainable_weights))
57              d_loss += current_d_loss / self.n_discriminator_updates
58
59          # Generate random latent vectors
60          random_latent_vectors = tf.random.normal(shape=(batch_size, self.
    latent_dim))
61
62          # Train generator
63          with tf.GradientTape() as tape:
```

```
            generated_images = self.generator(random_latent_vectors, training=True)
            fake_output = self.discriminator(generated_images, training=False)
            g_loss = self._generator_loss(fake_output)

        # Update generator weights
        grads = tape.gradient(g_loss, self.generator.trainable_weights)
        self.g_optimizer.apply_gradients(zip(grads, self.generator.trainable_weights))

        # Update loss trackers
        self.d_loss_tracker.update_state(d_loss)
        self.g_loss_tracker.update_state(g_loss)

        # Return loss metrics
        return {
            "d_loss": self.d_loss_tracker.result(),
            "g_loss": self.g_loss_tracker.result()
        }
```

22.4 Compilation

```
class GANMonitor(tfk.callbacks.Callback):

    # Initialise the callback with the number of images, latent dimension, name, and colour mode
    def __init__(self, num_img=10, latent_dim=latent_dim, name='', gray=False):
        self.num_img = num_img
        self.latent_dim = latent_dim
        self.name = name
        self.gray = gray

    # Generate and display images at the end of each epoch
    def on_epoch_end(self, epoch, logs=None):
        # Set random seed for reproducibility
        tf.random.set_seed(seed)

        # Ensure output directory exists
        os.makedirs(self.name + 'temp', exist_ok=True)

        # Generate latent vectors and images
        random_latent_vectors = tf.random.normal(shape=(self.num_img, self.latent_dim))
        generated_images = self.model.generator(random_latent_vectors).numpy()
        generated_images = generated_images * 0.5 + 0.5  # Rescale pixel values

        # Plot generated images
        fig, axes = plt.subplots(1, self.num_img, figsize=(20, self.num_img))
        for i in range(self.num_img):
            img = tfk.preprocessing.image.array_to_img(generated_images[i])
            ax = axes[i % self.num_img]
            if self.gray:
```

```
                ax.imshow(np.squeeze(img), cmap='gray')
            else:
                ax.imshow(np.squeeze(img))
            ax.axis('off')

        # Adjust layout and display the plot
        plt.tight_layout()
        plt.show()

# Create an instance of the GAN model with specified discriminator, generator, and latent dimension
gan = GAN(
    discriminator=get_discriminator(input_shape),
    generator=get_generator(latent_dim),
    latent_dim=latent_dim
)

# Compile the GAN model with Adam optimisers for both discriminator and generator
gan.compile(
    d_optimizer=tfk.optimizers.AdamW(learning_rate=learning_rate, beta_1=0.5, beta_2=0.999),
    g_optimizer=tfk.optimizers.Adam(learning_rate=learning_rate, beta_1=0.5, beta_2=0.999)
)

history = gan.fit(
    X,
    epochs=epochs,
    batch_size=batch_size,
    callbacks=[GANMonitor(name='fashionmnist_gan', gray=True)],
    verbose=2
).history
```

22.5 cGAN

We will build a cGAN based on the desired label:
We must change the input in both discriminator and generator:

```
# Define input layers for the image and conditioning label
input_layer_image = tfkl.Input(shape=input_shape, name='input_layer_image')
input_layer_conditioning = tfkl.Input(shape=(1,), name='input_layer_conditioning')

# Embed and process the conditioning input
x_c = tfkl.Embedding(num_classes, input_shape[0] * input_shape[1], name='embedding')(input_layer_conditioning)
x_c = tfkl.Flatten(name='flatten')(x_c)
x_c = tfkl.Reshape((input_shape[0], input_shape[1], 1))(x_c)

# Concatenate the conditioning information with the image input
x = tfkl.Concatenate(axis=-1)([input_layer_image, x_c])
```

Following a regular GAN and then changing the return model to

```
return tf.keras.Model(
    inputs=[input_layer_image, input_layer_conditioning],
```

```
    outputs=output_layer,
    name='conditional_discriminator/generator'
)
```

Then, in the GAN model object, we must change the discriminator and generator calls, using both regular input and conditioned label, for example:

```
generated_images = self.generator([random_latent_vectors, class_labels],
    training=True)
```

To compile it, we can call the function as regular GANs.
Finally, a conditional samples generator must be defined. It could be done as follows

```
def conditional_sample(model, num_img, latent_dim, num_classes=10, fixed=True
    , gray=False, label=None):
    # Optionally set a fixed random seed for reproducibility
    if fixed:
        tf.random.set_seed(seed)

    # Generate random latent vectors
    z = tf.random.normal(shape=(num_img, latent_dim))

    # Generate class labels or use the specified label
    if label is None:
        labels = tf.cast(tf.math.floormod(tf.range(0, num_img), num_classes),
     'int32')
    else:
        labels = tf.cast(tf.math.floormod(tf.ones(num_img) * label,
    num_classes), 'int32')

    # Reshape labels to match the expected input shape
    labels = tf.reshape(labels, (-1, 1))

    # Generate images using the model
    generated_images = model([z, labels]).numpy()

    # Display the generated images
    fig, axes = plt.subplots(1, num_img, figsize=(20, 2 * num_img))
    for i in range(num_img):
        img = tfk.preprocessing.image.array_to_img(generated_images[i])
        ax = axes[i % num_img]
        if gray:
            ax.imshow(np.squeeze(img), cmap='gray')
        else:
            ax.imshow(np.squeeze(img))
        ax.axis('off')
    plt.tight_layout()
    plt.show()

conditional_sample(cgan.generator, 10, latent_dim, gray=True)
```

23 Recurrent Neural Networks

23.1 Sequence Model

So far, we have considered only static datasets. There are different ways to deal with dynamic data, adding a time dimension:

- Memoryless Models (limited memory):
  - Autoregressive Models: Predict the next input from previous ones using "Delay taps" (delayed version of past inputs).
  - Feedforward Neural Networks: We can generalize autoregressive models using non-linear hidden layers, using delay taps.
- Models with Memory (Unlimited): Generative models with a hidden state that cannot be observed directly. The hidden states also propagate through one another, so they contain information about all the previous ones.
  - Linear Dynamic Systems
  - Hidden Markov Models
  - Recurrent Neural Networks

23.2 Recurrent Neural Networks

It adds memory via current connections. It adds some neurons to the layer $\{c_i^t\}_i$ that 'store memory' (dynamically changed by neurons ahead of them) when a forward pass occurs. These neurons that propagate behind them, are receiving information from both the input $\{x_i\}_i$ and store memory $\{c_i^t\}_i$, to propagate it behind to the $\{c_i^t\}_i$ and next layer (see figure 48).

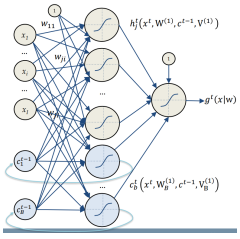


Figure 48: Recurrent Neural Network Basic architecture

Distributed hidden state allows to store information efficiently. Non-linear dynamics allows complex hidden state updates. The mathematic expressions for each neuron are the

following:

$$g^t(x_n|w) = g\left(\sum_{j=0}^{J} w_{1j}^{(2)} \cdot h_j^t(\cdot) + \sum_{b=0}^{B} v_{1b}^{(2)} \cdot c_b^t(\cdot)\right)$$
$$h_j^t(\cdot) = h_j\left(\sum_{i=0}^{I} w_{ji}^{(1)} \cdot x_{i,n}^t + \sum_{b=0}^{B} v_{jb}^{(1)} \cdot c_b^{t-1}\right)$$
$$c_b^t(\cdot) = c_b\left(\sum_{i=0}^{I} w_{bi}^{(1)} \cdot x_{i,n}^t + \sum_{b'=0}^{B} v_{bb'}^{(1)} \cdot c_{b'}^{t-1}\right)$$

Initial Memory states should be initialized. It could be done as a fixed value as 0, or even with learning parameters

23.3 Backpropagation Through Time

To deal with the loops in the memory, backpropagation through time creates copies of V_B and W_B (vectors of v_{jb} and w_{bi}'s). It performs this for U steps, being U an hyperparameter, and then it cuts the propagation.
The mathematic formula used is

$$W_B = W_B - \eta \cdot \frac{1}{U} \sum_{u=0}^{U-1} \frac{\partial E}{\partial W_B^{t-u}} V_B = V_B - \eta \cdot \frac{1}{U} \sum_{u=0}^{U-1} \frac{\partial E^t}{\partial V_B^{t-u}}$$

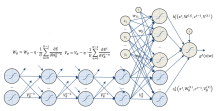


Figure 49: Backpropagation Through Time Representation

23.4 Vanishing Gradient

The vanishing gradient is a problem as we will see with the next example.
Consider the case of figure 50

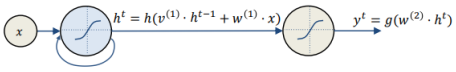


Figure 50: Vanishing Gradient in Recurrent Neural Networks

Backpropagation over an entire sequence S is computed as

$$\frac{\partial E}{\partial w} = \sum_{t=1}^{S} \frac{\partial E^t}{\partial w}$$
$$\frac{\partial E^t}{\partial w} = \sum_{t=1}^{t} \frac{\partial E^t}{\partial y^t}\frac{\partial y^t}{\partial h^t}\frac{\partial h^t}{\partial h^k}\frac{\partial h^k}{\partial w}$$
$$\frac{\partial h^t}{\partial h^k} = \prod_{i=k+1}^{t} \frac{\partial h_i}{\partial h_{i-1}} = \prod_{i=k+1}^{t} v^{(1)} h'(v^{(1)} \cdot h^{i-1} + w^{(1)} \cdot x)$$

So we see that the last expression affects the gradient.
If we consider the norm of the terms:

$$\|\frac{\partial h_i}{\partial h_{i-1}}\| \leq \|v^{(1)}\| \|h'(\cdot)\|$$
$$\implies \|\frac{\partial h^t}{\partial h^k}\| \leq (\gamma_v \cdot \gamma_{h'})^{t-k} \qquad \text{where } \gamma_x = \|x\|$$

So we can see that if $(\gamma_v \cdot \gamma_{h'}) < 1$, the gradient converges to 0. This means that if we use sigmoid and tanh, we have the vanishing gradient problem, since their derivatives are between -1 and 1.
A naive solution is using ReLu or the linear unit activation function. ReLu derivative can be 1 or 0. Linear unit derivative is always 1. The problem is that this would only accumulate the previous inputs, it won't learn complex patterns.

23.5 Long Short-Term Memory (LSTM)

Hochreiter and schmidhuber actually solved the problem in 1997, designing a memory cell using logistic and linear units by multiplicative interactions.

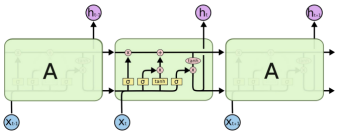


Figure 51: Long Short-Term Memory overall architecture

Using the following expressions:

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{1}$$
$$\tilde{C}_t = tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \tag{2}$$
$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{3}$$
$$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t \tag{4}$$
$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{5}$$
$$h_t = o_t \cdot tanh(C_t) \tag{6}$$

LSTM introduces three different components that interact with x_t (current input) and h_{t-1} (last period memory) using the sigmoid function as a gate, since it goes from 0 to 1:

1. **Forget gate:** It multiplies by weights and passes the components through a sigmoid function. Output values closer to 1 indicate that the input will not be forgotten. Output values closer to 0 indicate that the input will be forgotten (see equation 3).

2. **Input Gate:** It multiplies the results of multiplying the inputs by different weights and passing the components through a sigmoid and hyperbolic tangent function, respectively. The sigmoid decides what values we will update (see equation 1), while the tangent creates new candidates to update the memory (see equation 2).

3. **Connection input/forget:** The results of input and forget gates are added (see equation 4).

4. **Output gate:** It multiplies by weights, applying a sigmoid function (see equation 5). Then, it multiplies the result of it for the connection result applying a hyperbolic tangent (see equation 6).

23.6 Gated Recurrent Unit (GRU)

It combines the forget and input gates into a single update gate. It also merges the cell state and hidden state, and makes some other changes (see figure 52). It is more complex to understand, but it has fewer parameters.

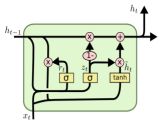


Figure 52: Gated Recurrent Unit Overall Architecture

$$
\begin{aligned}
z_t &= \sigma(W_z \cdot [h_{t-1}, x_t]) \\
r_t &= \sigma(W_r \cdot [h_{t-1}, x_t]) \\
\tilde{h}_t &= tanh(W \cdot [r_t \cdot h_{t-1}, x_t]) \\
h_t &= (1 - z_t) \cdot h_{t-1} + z_t \cdot \tilde{h}_t
\end{aligned}
$$

23.7 Multiple Layers and Bidirectional LSTM

Adding multiple layers allows us to create more complex representations.
There also exist biderectional LSTM, that allow transfer information left-to-right and right-to-left, concatenating hidden layers

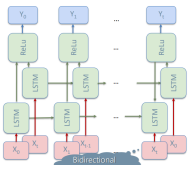


Figure 53: Bi Directional LSTM Architecture

23.8 Limitations

Despite the performance of these models being very good, it is difficult to train them largely

- Performing training and especially inference is sequential in nature.
- Parallelization at sample level is cut by recurrences
- Memory constraints limit batching across too many examples

This will be later solved by the use of transformers (see section 27.10)

24 Time Series Classification

This section has been created based on Prof. Eugenio Lomurno's notebook 1 and notebook 2

24.1 Import example

```
# Define column names for the dataset
column_names = ['user_id', 'activity', 'timestamp', 'x_axis', 'y_axis', '
    z_axis']
# Read the dataset into a DataFrame with specified column names
df = pd.read_csv('activities_recognition.txt', header=None, names=
    column_names)
# Remove rows with any missing values
df.dropna(axis=0, how='any', inplace=True)
```

24.2 First analyses

```
############## Count the number of unique users
n_users = len(df['user_id'].unique())
# Create a custom colour map with distinct colours for each user
colors = plt.cm.viridis(np.linspace(0, 1, n_users))
# Visualise the count of timestamps for each user
plt.figure(figsize=(17, 5))
sns.countplot(
    x='user_id',
    data=df,
    palette=colors
)
# Set the title of the plot
plt.title('Per User Timestamps')
# Display the plot
plt.show()

############## Identify unique activity executions per user by creating a
    composite ID
df['id'] = df['user_id'].astype('str') + '_' + df['activity'].astype('str')
# Print the number of unique activity executions
print(f'The dataset is composed of {df["id"].nunique()} different activity
    executions')
# Count the unique IDs for distinct activity executions
n_users = len(df['id'].unique())
```

```
# Create a custom colour map for better distinction of unique IDs
colors = plt.cm.turbo(np.linspace(0, 1, n_users))
# Visualise the count of timestamps per unique ID
plt.figure(figsize=(17, 5))
sns.countplot(
    x='id',
    data=df,
    order=df['id'].value_counts().index,
    palette=colors
)
# Set the title of the plot and disable x-axis labels for clarity
plt.title('Per Id Timestamps')
plt.xticks([], [])  # Remove x-axis ticks and labels
# Display the plot
plt.show()

############## Define a function to inspect sensor data for a specific
    activity
def inspect_activity(activity, df):
    # Filter the DataFrame for the specified activity and limit to 500 rows
    data = df[df['activity'] == activity][['x_axis', 'y_axis', 'z_axis'
    ]][:500]
    # Plot the sensor data for each axis
    axis = data.plot(subplots=True, figsize=(17, 9), title=activity)
    # Adjust legend position for each subplot
    for ax in axis:
        ax.legend(loc='lower right')
inspect_activity("Standing", df)
```

24.3 Time Series Classification

24.3.1 Cross Validation

```
# Split the dataset into training, validation, and test sets based on user ID
df_train = df[df['user_id'] <= 26]  # Training set: user IDs 1 to 26
df_val = df[(df['user_id'] > 26) & (df['user_id'] <= 31)]  # Validation set:
    user IDs 27 to 31
df_test = df[df['user_id'] > 31]  # Test set: user IDs 32 and above

# Print the shapes of the training, validation, and test sets
df_train.shape, df_val.shape, df_test.shape
```

24.3.2 Preprocessing

```
############## Normalize columns and map labels to numbers


############## Define a function to build sequences from the dataset. (We need
     a fixed size)
def build_sequences(df, window=200, stride=200):
    # Sanity check to ensure the window is divisible by the stride
    assert window % stride == 0

    # Initialise lists to store sequences and their corresponding labels
    dataset = []
```

```
    labels = []

    # Iterate over unique IDs in the DataFrame
    for id in df['id'].unique():
        # Extract sensor data for the current ID
        temp = df[df['id'] == id][['x_axis', 'y_axis', 'z_axis']].values

        # Retrieve the activity label for the current ID
        label = df[df['id'] == id]['activity'].values[0]

        # Calculate padding length to ensure full windows
        padding_len = window - len(temp) % window

        # Create zero padding and concatenate with the data
        padding = np.zeros((padding_len, 3), dtype='float32')
        temp = np.concatenate((temp, padding))

        # Build feature windows and associate them with labels
        idx = 0
        while idx + window <= len(temp):
            dataset.append(temp[idx:idx + window])
            labels.append(label)
            idx += stride

    # Convert lists to numpy arrays for further processing
    dataset = np.array(dataset)
    labels = np.array(labels)
    return dataset, labels
############## Generate sequences
# Generate sequences and labels for the training set
X_train, y_train = build_sequences(df_train, window, stride)
# Generate sequences and labels for the validation set
X_val, y_val = build_sequences(df_val, window, stride)
# Generate sequences and labels for the test set
X_test, y_test = build_sequences(df_test, window, stride)
# Print the shapes of the generated datasets and their labels
X_train.shape, y_train.shape, X_val.shape, y_val.shape, X_test.shape, y_test.
    shape
# Generate sequences and labels for the training set
X_train, y_train = build_sequences(df_train, window, stride)
# Generate sequences and labels for the validation set
X_val, y_val = build_sequences(df_val, window, stride)
# Generate sequences and labels for the test set
X_test, y_test = build_sequences(df_test, window, stride)
# Print the shapes of the generated datasets and their labels
X_train.shape, y_train.shape, X_val.shape, y_val.shape, X_test.shape, y_test.
    shape
```

24.3.3 LSTM

The LSTM block can be built with the follow command

```
x = tfkl.LSTM(128, return_sequences=True, name='lstm_0')(x)
```

So an LSTM network can be built as follows

```
# Define a function to build an LSTM-based classifier
```

```
def build_LSTM_classifier(input_shape, classes):
    # Define the input layer
    input_layer = tfkl.Input(shape=input_shape, name='Input')

    # Add feature extractor layers
    x = tfkl.LSTM(128, return_sequences=True, name='lstm_0')(input_layer)
    x = tfkl.BatchNormalization(name='batch_norm_0')(x)

    x = tfkl.LSTM(128, return_sequences=True, name='lstm_1')(x)
    x = tfkl.BatchNormalization(name='batch_norm_1')(x)

    x = tfkl.LSTM(128, name='lstm_2')(x)
    x = tfkl.BatchNormalization(name='batch_norm_2')(x)

    # Add dropout for regularisation
    x = tfkl.Dropout(0.5, name='dropout')(x)

    # Add classifier layers
    x = tfkl.Dense(128, name='dense_hidden')(x)
    x = tfkl.BatchNormalization(name='dense_hidden_batch_norm')(x)
    x = tfkl.Activation('relu', name='dense_hidden_activation')(x)

    x = tfkl.Dense(classes, name='dense_output')(x)
    output_layer = tfkl.Activation('softmax', name='dense_output_activation')
    (x)

    # Create the model by connecting input and output layers
    model = tfk.Model(inputs=input_layer, outputs=output_layer, name='model')

    # Compile the model with categorical crossentropy loss and Adam optimiser
    model.compile(
        loss=tfk.losses.CategoricalCrossentropy(),
        optimizer=tfk.optimizers.Adam(),
        metrics=['accuracy']
    )

    # Return the compiled model
    return model
```

Then we have to train the model as always
We can also build bidirectional LSTMs. This can be done with the Bidirectional function, so the whole block would be built as follows:

```
    x = tfkl.Bidirectional(tfkl.LSTM(128, return_sequences=True), name='
    bilstm_0')(x)
```

24.3.4 1D convolution

A more traditional approach can be also tried, using convolutions (with its 1D version, since are vectors)

```
# Define a function to build a 1D CNN-based classifier
def build_1DCNN_classifier(input_shape, classes):
    # Define the input layer
    input_layer = tfkl.Input(shape=input_shape, name='Input')

    # Add feature extractor layers
```

```
    x = tfkl.Conv1D(128, 3, padding='same', name='conv1d_0')(input_layer)
    x = tfkl.BatchNormalization(name='batch_norm_0')(x)
    x = tfkl.Activation('relu', name='activation_0')(x)
    x = tfkl.MaxPooling1D(name='max_pooling_0')(x)

    x = tfkl.Conv1D(128, 3, padding='same', name='conv1d_1')(x)
    x = tfkl.BatchNormalization(name='batch_norm_1')(x)
    x = tfkl.Activation('relu', name='activation_1')(x)
    x = tfkl.MaxPooling1D(name='max_pooling_1')(x)

    x = tfkl.Conv1D(128, 3, padding='same', name='conv1d_2')(x)
    x = tfkl.BatchNormalization(name='batch_norm_2')(x)
    x = tfkl.Activation('relu', name='activation_2')(x)
    x = tfkl.GlobalAveragePooling1D(name='gap')(x)

    # Add dropout for regularisation
    x = tfkl.Dropout(0.5, name='dropout')(x)

    # Add classifier layers
    x = tfkl.Dense(128, name='dense_hidden')(x)
    x = tfkl.BatchNormalization(name='dense_hidden_batch_norm')(x)
    x = tfkl.Activation('relu', name='dense_hidden_activation')(x)

    x = tfkl.Dense(classes, name='dense_output')(x)
    output_layer = tfkl.Activation('softmax', name='dense_output_activation')
    (x)

    # Create the model by connecting input and output layers
    model = tfk.Model(inputs=input_layer, outputs=output_layer, name='model')

    # Compile the model with categorical crossentropy loss and Adam optimiser
    model.compile(
        loss=tfk.losses.CategoricalCrossentropy(),
        optimizer=tfk.optimizers.Adam(),
        metrics=['accuracy']
    )

    # Return the compiled model
    return model
```

Even if the architecture is not specifically designed for time relations, it is actually able to extract patrons anyway.

24.4 Time Series Forecasting

24.4.1 Cross Validation

```
X_train_raw = dataset.iloc[:-val_size-test_size]
X_val_raw = dataset.iloc[-val_size-test_size:-test_size]
X_test_raw = dataset.iloc[-test_size:]
```

24.4.2 Preprocessing

```
# Define window size for time-series data
window = 50
```

```
# Define stride length for window movement
stride = 5
# Define telescope size for prediction horizon
telescope = 50

# Define a function to build sequences and corresponding labels from a
    dataframe
def build_sequences(df, target_labels=['temperature'], window=200, stride=20,
     telescope=100):
    # Ensure the window size is compatible with the stride
    assert window % stride == 0

    # Initialise containers for dataset sequences and labels
    dataset = []
    labels = []

    # Copy dataframe values for processing
    temp_df = df.copy().values
    temp_label = df[target_labels].copy().values

    # Check for and handle padding requirement
    padding_check = len(df) % window
    if padding_check != 0:
        # Compute padding length
        padding_len = window - len(df) % window

        # Add zero-padding to features and labels
        padding = np.zeros((padding_len, temp_df.shape[1]), dtype='float32')
        temp_df = np.concatenate((padding, temp_df))

        padding = np.zeros((padding_len, temp_label.shape[1]), dtype='float32
    ')
        temp_label = np.concatenate((padding, temp_label))

        # Verify that the padded data length is divisible by the window size
        assert len(temp_df) % window == 0

    # Create sequences and corresponding labels
    for idx in np.arange(0, len(temp_df) - window - telescope, stride):
        dataset.append(temp_df[idx:idx + window])
        labels.append(temp_label[idx + window:idx + window + telescope])

    # Convert lists to numpy arrays
    dataset = np.array(dataset)
    labels = np.array(labels)

    # Return the dataset and labels
    return dataset, labels
```

If we want to create an autoregressive model, we have to use a telescope smaller than the prediction window, so the model will use the predictions as input in the next prediction

24.4.3 Convolutional LSTM model

```
def build_CONV_LSTM_model(input_shape, output_shape):
```

```
    # Ensure the input time steps are at least as many as the output time steps
    assert input_shape[0] >= output_shape[0], "For this exercise we want input time steps to be >= of output time steps"

    # Define the input layer with the specified shape
    input_layer = tfkl.Input(shape=input_shape, name='input_layer')

    # Add a first LSTM layer with 128 units
    x = tfkl.LSTM(128, return_sequences=True, name='lstm1')(input_layer)
    x = tfkl.Dropout(0.3)(x)

    # Add a second LSTM layer with 128 units
    x = tfkl.LSTM(128, return_sequences=True, name='lstm2')(x)
    x = tfkl.Dropout(0.3)(x)

    # Add a 1D Convolution layer with 128 filters and a kernel size of 3
    x = tfkl.Conv1D(128, 3, padding='same', name='conv1')(x)
    x = tfkl.Activation('relu', name='relu_after_conv1')(x)
    x = tfkl.Dropout(0.3)(x)

    # Add a 1D Convolution layer with 128 filters and a kernel size of 3
    x = tfkl.Conv1D(128, 3, padding='same', name='conv2')(x)
    x = tfkl.Activation('relu', name='relu_after_conv2')(x)
    x = tfkl.Dropout(0.3)(x)

    # Add a final Convolution layer to match the desired output shape
    output_layer = tfkl.Conv1D(output_shape[1], 3, padding='same', name='output_layer')(x)

    # Calculate the size to crop from the output to match the output shape
    crop_size = output_layer.shape[1] - output_shape[0]

    # Crop the output to the desired length
    output_layer = tfkl.Cropping1D((0, crop_size), name='cropping')(output_layer)

    # Construct the model by connecting input and output layers
    model = tf.keras.Model(inputs=input_layer, outputs=output_layer, name='CONV_LSTM_model')

    # Compile the model with Mean Squared Error loss and Adam optimizer
    model.compile(loss=tf.keras.losses.MeanSquaredError(), optimizer=tf.keras.optimizers.AdamW())

    return model
```

24.4.4 Prediction

Therefore we can use the trained model to predict the future

```
predictions = model.predict(X_test, verbose=0)

# Compute Mean Absolute Errors (MAEs) for each feature and prediction step
maes = []

# Loop over each timestep in the predictions
```

```
for i in range(predictions.shape[1]):
    ft_maes = []

    # Loop over each feature in the predictions
    for j in range(predictions.shape[2]):
        # Calculate MAE for the current feature and timestep
        ft_maes.append(np.mean(np.abs(y_test[:, i, j] - predictions[:, i, j])
    , axis=0))

    # Convert feature MAEs to a numpy array and append to overall MAEs
    ft_maes = np.array(ft_maes)
    maes.append(ft_maes)

# Convert MAEs to a numpy array
maes = np.array(maes)

# Generate predictions for future data using the trained model
future_predictions = model.predict(future, verbose=0)

# Concatenate the last known point of the future data with the predictions
future_predictions = np.concatenate([np.expand_dims(future[:, -1, :], axis=0)
    , future_predictions], axis=1)

# Append zeros to the beginning of the MAEs array for alignment
maes = np.concatenate([np.array([[0, 0, 0]]), maes], axis=0)
```

24.5 Autoregressive Prediction

```
# Perform autoregressive forecasting
reg_predictions = np.array([])  # Initialise an empty array to store
    predictions
X_temp = X_test_reg  # Temporary variable to store input data for
    autoregressive steps

# Iterate through the telescope in steps of the autoregressive telescope size
for reg in range(0, telescope, autoregressive_telescope):
    # Predict the next set of values using the model
    pred_temp = model.predict(X_temp, verbose=0)

    # Concatenate predictions to the overall results
    if len(reg_predictions) == 0:
        reg_predictions = pred_temp
    else:
        reg_predictions = np.concatenate((reg_predictions, pred_temp), axis
    =1)

    # Update the input data for the next prediction step by appending the
    predictions
    X_temp = np.concatenate((X_temp[:, autoregressive_telescope:, :],
    pred_temp), axis=1)
```

25 Sequence to Sequence

25.1 Sequential Data Problems

- **One to One:** Fixed-size input to a fixed size output
- **One to Many:** Sequence output. E.g.: Image Captioning
- **Many to One:** Sequence input. E.g.: Sentiment Classification Analysis of a phrase
- **Many to Many:** Sequence input and sequence output. E.g.: Language Translation
- **Many to Many:** Synchronized sequence input and output

25.2 Conditional Language Models

Language Models represent the probability of a sequence, for instance, a sentence.

$$P(y_1, y_2, \ldots, y_n) = \prod_{t=1}^{n} p(y_t | y_{<t})$$

Conditional language models condition on a source sentence

$$P(y_1, y_2, \ldots, y_n | x_1, x_2, \ldots, x_m) = \prod_{t=1}^{n} p(y_t | y_{<t}, x_1, x_2, \ldots, x_m)$$

In the case of image captioning $x_1, x_2, \ldots, x_m$ can be replaced by an image x

25.3 Sequence basics

Given an input sentence $x_1, x_2, \ldots, x_m$ and a target output sequence $y_1, y_2, \ldots, y_n$, we aim the sequence which maximizes the conditional probability $P(y|x)$

$$y^* = argmax_y P(y_1, y_2, \ldots, y_n | x_1, x_2, \ldots, x_m)$$

In sequence to sequence modeling, we learn from data a model $P(y|x, \theta)$, and our prediction becomes

$$y^* = argmax_y P(y_1, y_2, \ldots, y_n | x_1, x_2, \ldots, x_m, \theta)$$

25.4 Encoder-Decoder architecture

Sequence to sequence models are usually built over encoder-decoder architectures. It builds an abstract representation and then decodes it into a new sequence

25.5 Greedy Decoding vs Beam Search

Once the model is trained, each sequence is predicted using

$$y' = argmax_y \prod_{t=1}^{n} p(y_t | y_{<t}, x_1, x_2, \ldots, x_m, \theta)$$

To compute the argmax over all possible sequences we can use.

- **Greedy Decoding:** At each step, picking the most probable token

$$y' = argmax_y \prod_{t=1}^{n} p(y_t|y_{<t}, x_1, x_2, \ldots, x_m, \theta) \approx \prod_{t=1}^{n} argmax_y p(y_t|y_{<t}, x_1, x_2, \ldots, x_m, \theta)$$

  This does not guarantee reaching the best sequence and does not allow us to backtrack from errors in early classification stages. The advantage is that the algorithm complexity is linear

- **Beam Search:** Keep track of several most probable hypotheses, keeping a fixed number of most probable predictions in each step. In the intermediate steps, in each level the algorithm drops the $n - r$ less probable sequences, reducing the algorithm's complexity to linear $r \cdot m$.

25.6 Model Training

25.6.1 The Loss

At time t, the model predicts

$$p_t = p(\cdot|y_1, y_2, \ldots, y_{t-1}, x_1, x_2, \ldots, x_m)$$

Using a one-hot vector for y_t we can use the cross entropy as a loss

$$loss_t(p_t, y_t) = -\sum_{i=1}^{|V|} y_t^{(i)} log(p_t^{(i)}) = -y_t^T log(p_t)$$

Then, over the entire sequence:

$$loss = -\sum_{t=1}^{n} y_t^T log(p_t)$$

During the training, the model can calculate this loss every time it generates a new token. The next token could be shown just after the model predicts it, so the algorithm can calculate the loss and update the weights.

The problem with this approach is that it is slow, since each step has to wait for the previous one. A better approach is to consider predicting each next token based on the previous ones (using inputs and labels) as an independent task (see figure 54). This makes this algorithm parallelizable.

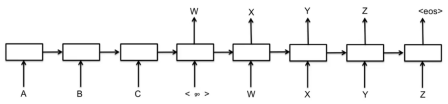


Figure 54: Training process in Sequence to Sequence Models

On the other hand, at inference time, the time decoder feeds the output of each t as input of $t + 1$. Since we don't have labels, it is not parallelizable as training time.

25.7 Dataset Preparation

There are special characters that can be predicted, to solve problems during the handle of the training:

- $< PAD >$**:** During the training, examples are given to the network in batches. The inputs in these batches need to be the same long. This character is used to pad shorter inputs to the same width of the batch.
- $< EOS >$**:** It indicates that the sentence ends. This means that we have to stop to predict the next character.
- $< UNK >$**:** On real data, it can vastly improve the resource efficiency to ignore words that do not often appear in your vocabulary by replacing those with this character.
- $< SOS > / < GO >$**:** This is the input to the first-time step of the decoder to let the decoder know when to start generating output.

Using these characters, we have to prepare the dataset as follows:

1. Sample batch_size pairs of (source_sequence, target_sequence)
2. Append $< EOS >$ at the end of the source_sequence
3. Add $< SOS >$ at the beginning of the target_sequence to obtain target_input_sequence.
4. Append $< EOS >$ at the end of the target_sequence to obtain target_output_sequence.
5. Pad up to the max_input_length within the batch using $< PAD >$
6. Encode tokens based on vocabulary or embedding, including vocabulary and these special characters (see section 26)
7. Replace out-of-vocabulary (OOV) tokens with $< UNK >$.

26 Word Embedding

26.1 Introduction

Natural Language Processing treats words as discrete atomic symbols, as items in a dictionary. It means that if we want to create a vector that indicates if a word is in a sentence, the dimensionality of it would be as long as words in the vocabulary. Plus, the vectors would be sparse, since most of the words would not be in most of the sentences.

26.2 N-Grams

Determine $P(s_k = w_1, \ldots, w_k)$ in some domain of interest

$$P(s_k) = \prod_{i}^{k} P(w_i | w_1, \ldots, w_{i-1})$$

In traditional n-gram language models, "the probability of a word depends only on the context of n-1 previous words"

$$\hat{P}(s_k) = \prod_i^k P(w_i|w_{i-n+1}, \ldots, w_{i-1})$$

So the typical ML-smoothing learning process is

1. $P(w_i|w_{i-n+1}, \ldots, w_{i-1}) = \frac{\#w_{i-n+1},\ldots,w_{i-1},w_i}{\#w_{i-n+1},\ldots,w_{i-1}}$
2. Smooth to avoid zero probability by assigning probability to unseen sequences

So if we use a 10-gram language model on a corpus of 100000 unique words. The model lives in a 10D hypercube, where each dimension has 100000 slots.

Therefore, the model training consists of assigning a probability of each of the 100000^{10} slots.

Probability mass vanishes, so more data is needed to fill the huge space.

The more data, the more unique words, so the space grows. In practice, corpora can have 10^6 unique words.

Contexts in N-gram are usually limited to size 2, called trigram model. This does not allow us to capture a lot of context

The perpendicularity of similar sentences with different words suggests that we are not representing the text efficiently.

26.3 Embedding

We call embedding to any technique mapping a word or phrase from its original high-dimensional input space to a lower-dimensional numerical vector space, where closer points are instances that are closer in meaning.

Therefore, we can use the architecture of autoencoders (see section 19) to reduce the dimensionality of the original space.
We can add a sparsity term to the loss as follows:

$$E = \underbrace{\|g_i(x_i|w) - x_i\|^2}_{Reconstruction\ Error} + \underbrace{\lambda \sum_j \left| h_j(\sum_i w_{ji}^{(1)} x_i) \right|}_{Sparsity\ term}$$

Doing this, we map each unique word in a vocabulary V ($\|V\| > 10^6$) to a continuous m-dimensional space, with typically $100 < m < 500$

26.4 Models developed

26.4.1 Neural Net Language Model

It was an architecture developed by Bengio et.al. in 2003.
The model's goal is to use a (n-1)-gram model to embed the words in m layers, using a vocabulary with a size of $|V|$.

It starts setting the input to the one hot encoding representation of the words in the vocabulary. Then, it projects the words to a $C_{|V|\times m}$ matrix, from the one hot representation

(V-dimensional) to the embedding space (m-dimensional). This results in a $n \times m$ projection in the embedding space. Sequentially, it applies some hidden layers (initially neurons with a single hidden layer) with non-linear transformations (originally a tanh) to finally have a softmax output layer.

The complexity is $n \times m + n \times m \times h + h \times |V|$, kind of costly, even if it works very well. The main Bengio's contribution was uncovering the possibility of embedding the words in smaller dimensions.

26.4.2 Word2vec

The 2 principal models were developed by Google in 2013, to achieve a better performance model. It deletes the hidden layer, makes the projection layer shared, and changes the context to past and future.

Google developed something called Continuous Bag-of-Words (CBOW) architecture. It consists of the same as Bengio's model, but after the projection layer, it calculates the average of the words in the context, remaining only m neurons for the whole phrase. Then, it passes directly to the output, without any hidden layer.

The complexity this time went down to $n \times m + m \times \log |V|$ (the log is due to the use of Hierarchical Softmax, a method that approximated it efficiently).

This model does not use biases in the embeddings, it is just a matrix multiplication, but it does in the output layer.

26.5 Regularities in word2vec embedding space

It has been discovered some regularities in the embedding spaces in word2vec. There are some constant vectors that approximate the relation between concepts, like gender or country-capital. This allows us to operate over the vectors to approximate words that we didn't get in the first place.

27 Attention Mechanism

27.1 Introduction

The attention in the encoder-decoder architecture consists of deciding which parts of the source are more important. In addition to the bottleneck information, the decoder also examines the encoder's different chain stages, using a softmax to decide whether a representation is relevant to the sequence construction so that it can be used.

27.2 Mechanism

Let be $s_1, s_2, \ldots, s_m$ all the encoder states. Let h_t a decoder state in the step t. The attention algorithm calculates how relevant is a source token k for target step t.

$$score_{t,k} = score(h_t, s_k) \; \forall k = 1, \ldots, m$$

It uses these scores to calculate the attention weight for source token k at decoder step t (using the softmax function).

$$a_k^{(t)} = \frac{\exp(score_{t,k})}{\sum_{i=1}^{m} \exp(score_{t,i})} \ \forall k = 1, \ldots, m$$

So finally, the algorithm can calculate the attention output, the weighted sum of the weights times the encoder states.

$$c^{(t)} = a_1^{(t)} s_1 + a_2^{(t)} s_2 + \cdots + a_m^{(t)} s_m = \sum_{k=1}^{m} a_k^{(t)} s_k$$

The layer uses this output to concatenate it or sum it to the rest of the classic architecture.

27.3 Attention Scores

There are different mechanisms to compute attention scores from less to more complexity:

- Simple dot-product
$$score_{t,k} = h_t^T s_k$$
- Bilinear function aka 'Luong attention'
$$score_{t,k} = h_t^T W s_k$$
- Multi-layer perceptron
$$score_{t,k} = w_2^T \cdot tanh(W_1[h_t, s_k])$$
- 'Bahdanau attention'
Like a multi-layer perceptron, but it concatenates states from forward and backward RNNs

Note that the last ones add more parameters with W

27.4 Chatbots

This architecture can be applied to a conversation between two agents. In this situation, we can create chatbots.
Chatbots can be defined in at least two ways:

- **Generative (Core algorithm):** Encode the question into a context vector and generate the answer word by word using conditioned probability over answer vocabulary.
- **Retrieval (Context handling):** Rely on knowledge base of question-answer pairs. When a new question comes in, the inference phase encodes it in a context vector and, by using similarity measure, retrieves the top-k neighbor knowledge base items. It returns the most probable answer in the training set.

Also, the memory can be handled in two different ways:

- **Single-turn:** Build the input vector by considering just 1 question

$$\{(q_i, a_i)\}$$

- **Multi-turn:** Build the input vector by considering multi-turn conversational context.

$$\{([q_{i-2}; a_{i-2}; q_{i-1}; a_{i-1}; q_i], a_i)\}$$

27.5 Hierarchical Chatbots

LSTM cells often miss long-term dependencies within input sequences that are longer than 100 tokens.

Xing et. al., in 2017, extended the attention mechanism from single-turn response generation to a hierarchical attention mechanism. It generates hidden representations of sentences from contextualized words. Then it uses not only characters and words to put attention on, but also sentences, allowing having way longer context, focusing on the important part of the phrases only.

27.6 Self-Attention

The self-attention mechanism is done by the Query, Key, and Value mechanism, which performs a matrix multiplication with W_Q, W_K, and W_V to obtain the query, key, and value vectors, respectively. The query is the vector from which the attention is looking. The key says what information the token provides. The value is the actual information of the token.

The token we want to pass through in the layer is transformed into the query, and all the other ones are transformed to both key and value representation. For each token, the key is performing attention with the query. Finally, the attention weights are applied to the values.
The Attention is calculated as follows:

$$softmax\left(\frac{qk^T}{\sqrt{d_k}}\right)v$$

where d_k is the vector dimensionality of k.

This architecture allows us to calculate in parallel all the tokens in a fixed layer.

27.7 Multi-head Attention

Attention defines the role of a word in a sentence. This, in turn, might be related to different aspects, such as gender, number, etc. Multiple-head attention allows the model to focus on different things, both at encoding and decoding time.
This is implemented as the concatenation of several attention heads:

$$Multihead(Q, k, V) = Concat(Head_1, Head_2, \ldots, Head_n)W_0$$

$$Head_i = Attention(QW_Q^i, VW_V^i, KW_K^i)$$

27.8 Masked Self-Attention

In the decoder, the attention mechanism works differently at training and inference time. In inference, we cannot look ahead. However, at training time, we can, so we might want to process it in parallel ('look ahead' problem).

This dissonance is solved by masking future tokens, adding a mask matrix in the numerator of the softmax, adding -inf to the elements we should not look at in training time

27.9 Positional Embeddings

The self-attention mechanism is permutation invariant since it does not depend on the position nor the order of words in the sequence.

An additional feature included was the positional encoding applied to input and output embeddings. The positional encoding is used to make self-attention depending also on the position of the input.

A token input representation (what we will feed the network in section 27.10 with) is the sum of two embeddings:

1. Tokens
2. Positions

Positional embeddings can be learned, but transformers use fixed positional encodings:

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{model}})$$

$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{model}})$$

where:

- pos is the position
- i is the vector dimension
- d_{model} is the input size

Anyway, positional encoding can be used with other techniques:

- Concatenating instead of adding
- Being parameters of the network

27.10 Transformers (Attention is all you need)

27.10.1 Introduction

In 2017, Google Brain proposed the Transformer model to speed up training by replacing the sequential RNN architecture with an attention mechanism that could be parallelized. Specifically, attention is used not only in the connections between encoder-decoder, but also within the internal architecture of both the encoder and decoder.

Transformers use an encoder-decoder-like architecture, but with a different focus. The encoder is in charge of taking the source sequence and giving a context-aware representation

of the words, keeping constant the dimension of the sequence. On the other hand, the decoder is in charge of taking the output of the encoder, and the input target sequence. It has to predict the next element with respect to the input target sequence, using its inputs.

27.10.2 Feeding Encoder and Decoder

Both the encoder and decoder are not fed with the raw word representations. They are fed with embeddings of the source and input target sequence, respectively. These embeddings are obtained by the embedding layer. The embedding layer used can be a pre-trained layer or a layer trained on the transformer training. In this second case, it is trained just before the encoder, although is used in both encoder and decoder.

Beyond that, the real input consists of the corresponding embeddings added with the positional encoding seen in section 27.9. Each element in the sequence input is just the sum of the embedding plus the input embedding.

27.10.3 Encoder

At the encoder, the self-attention (see section 27.6) operates a different representation of the input token i, taking attention to every token of the same layer to predict the new representations of token i. This can be done in parallel, since we are using only the fixed previous layer tokens to predict the ones in the current layer.

So, in practice, self-attention looks at the sequence all at once. The output of each block is the sum of the output of multiple-self-attention (see section 27.7) added with a skip connection that is used for facilitating the information passing and avoiding the vanishing gradient.

Finally, a layer normalization is done.

27.10.4 Decoder

In the decoder, the self-attention receives the encoder output concatenated with the target tokens, masked as seen in section 27.8. It follows the same self-attention as the encoder, but with this sequence.

The decoder training is done in parallel, but the prediction must be sequential, since we do not have the targets a priori.

27.10.5 Encoder-Decoder Attention

Plus, each layer in the decoder is connected by an encoder-decoder attention. It uses the query-key-value technique, but using queries from the decoder, and keys and values from the encoder. This allows the decoder to extract valuable information from the encoder.

27.10.6 Feed Forward Layers

Finally, after the decoder, a feed-forward network takes the output in order to process the information. In any case, it is added a skip connection with an add operation directly to the final, where a normalization layer, linear activation, and a softmax are applied sequentially.

27.10.7 Overall representation

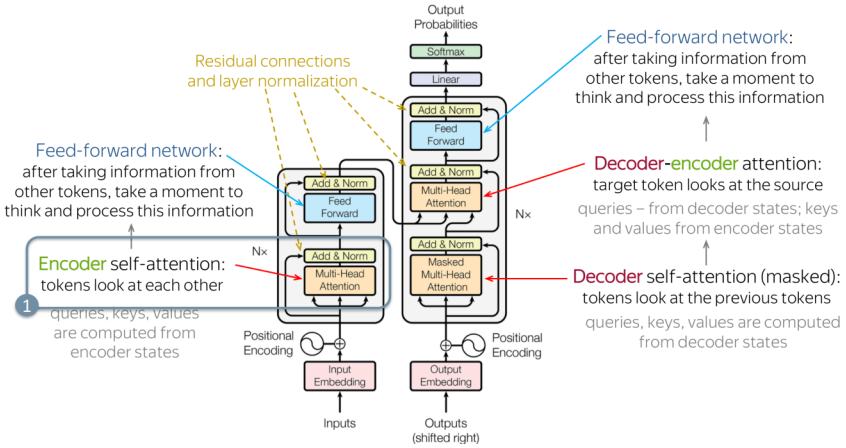


Figure 55: Basic Transformer Architecture

28 Transformers Implementation

This section has been created based on Prof. Eugenio Lomurno's notebook 1 and notebook 2

28.1 Preamble: Attention

28.1.1 Scaled Dot-Product Attention

```
# Define cross-attention layer with scaling and dot-product score mode
cross_attn = tf.keras.layers.Attention(use_scale=True, score_mode='dot')
Cross Attention
```

```
# Compute cross-attention output with Q as query, and K, V from another
    sequence
cross_output = cross_attn([Q, V, K])
```

Self-Attention

```
# Compute self-attention output with Q used as query, keys, and values
self_output = self_attn([Q, Q, Q])
```

28.1.2 Muti-head attention

```
# Define number of attention heads and compute head dimension
num_heads = 4
head_dim = d_model // num_heads # Convention
# Define a multi-head attention layer with 2 heads and specified key
    dimension
mha = tf.keras.layers.MultiHeadAttention(num_heads=2, key_dim=head_dim)

# Compute multi-head attention output and attention scores
output, attention_heads = mha(query=Q, value=V, key=K,
    return_attention_scores=True)
```

If we want to do it with self-attention we just put Q as a query, value, and key

28.2 Data Preprocessing

```
# Data processing parameters
vocab_size = 10000
sequence_length = 16
batch_size = 512

# Model architecture dimensions
embed_dim = 256
latent_dim = 512
num_heads = 4

# Layer configuration
num_encoder_layers = 2
num_decoder_layers = 2

#### English-Italian translation dataset
train_pairs = text_pairs[:num_train_samples]
val_pairs = text_pairs[num_train_samples:num_train_samples + num_val_samples]
test_pairs = text_pairs[num_train_samples + num_val_samples:]


# Define character set for stripping
strip_chars = string.punctuation + "  "
strip_chars = strip_chars.replace("[", "")
strip_chars = strip_chars.replace("]", "")

# Define text standardisation function
def custom_standardization(input_string):
    lowercase = tf.strings.lower(input_string)
    return tf.strings.regex_replace(lowercase, "[%s]" % re.escape(strip_chars
    ), "")

# Configure vectorisation layers
eng_vectorization = tfkl.TextVectorization(
    max_tokens=vocab_size,
    output_mode="int",
    output_sequence_length=sequence_length,
)
ita_vectorization = tfkl.TextVectorization(
    max_tokens=vocab_size,
    output_mode="int",
    output_sequence_length=sequence_length + 1,
    standardize=custom_standardization,
)

# Extract texts and adapt vectors
train_eng_texts = [pair[0] for pair in train_pairs]
train_ita_texts = [pair[1] for pair in train_pairs]
eng_vectorization.adapt(train_eng_texts)
ita_vectorization.adapt(train_ita_texts)


def format_dataset(eng, ita):
```

```
    # Vectorise input texts
    eng = eng_vectorization(eng)
    ita = ita_vectorization(ita)

    # Structure inputs and targets
    return (
        {
            "encoder_inputs": eng,
            "decoder_inputs": ita[:, :-1],
        },
        ita[:, 1:],
    )


def make_dataset(pairs):
    # Extract and convert text pairs
    eng_texts, ita_texts = zip(*pairs)
    eng_texts = list(eng_texts)
    ita_texts = list(ita_texts)

    # Create and configure dataset
    dataset = tf.data.Dataset.from_tensor_slices((eng_texts, ita_texts))
    dataset = dataset.batch(batch_size)
    dataset = dataset.map(format_dataset)

    return dataset.cache().shuffle(2048).prefetch(16)


# Create train, validation and test datasets
train_ds = make_dataset(train_pairs)
val_ds = make_dataset(val_pairs)
test_ds = make_dataset(test_pairs)

# Print tensor shapes from sample batch
for inputs, targets in train_ds.take(1):
    print(f"Tensor shapes from sample batch:\n"
          f"  Encoder inputs:  {inputs['encoder_inputs'].shape}\n"
          f"  Decoder inputs:  {inputs['decoder_inputs'].shape}\n"
          f"  Target outputs:  {targets.shape}")
```

28.3 Architecture

Positional Embedding Block

```
@keras.saving.register_keras_serializable(package="TransformerComponents")
class PositionalEmbedding(tfkl.Layer):
    def __init__(self, sequence_length, vocab_size, embed_dim, **kwargs):
        # Initialise parent class
        super().__init__(**kwargs)

        # Configure token embedding layer
        self.token_embeddings = tfkl.Embedding(
            input_dim=vocab_size,
            output_dim=embed_dim,
            mask_zero=True  # Enable automatic mask handling
        )
```

```
13
14          # Store layer parameters
15          self.sequence_length = sequence_length
16          self.vocab_size = vocab_size
17          self.embed_dim = embed_dim
18
19          # Generate positional encoding matrix
20          self.pos_encoding = self.get_positional_encoding(sequence_length,
    embed_dim)
21
22      def get_positional_encoding(self, position, d_model):
23          # Calculate position encoding as per transformer paper
24          pos = np.arange(position)[:, np.newaxis]
25          div_term = np.exp(np.arange(0, d_model, 2) * (-np.log(10000.0) /
    d_model))
26
27          # Compute sinusoidal encoding components
28          pe = np.zeros((position, d_model))
29          pe[:, 0::2] = np.sin(pos * div_term)
30          pe[:, 1::2] = np.cos(pos * div_term)
31
32          return tf.cast(pe[np.newaxis, ...], dtype=tf.float32)
33
34      def call(self, inputs):
35          # Extract sequence length and embed tokens
36          length = tf.shape(inputs)[-1]
37          embedded_tokens = self.token_embeddings(inputs)
38
39          # Apply positional encoding for current sequence length
40          positional_encoding = self.pos_encoding[:, :length, :]
41
42          return embedded_tokens + positional_encoding
43
44      def get_config(self):
45          # Enable layer serialisation
46          config = super().get_config()
47          config.update({
48              "sequence_length": self.sequence_length,
49              "vocab_size": self.vocab_size,
50              "embed_dim": self.embed_dim,
51          })
52          return config
```

Transformer Encoder Block

```
1  @keras.saving.register_keras_serializable(package="TransformerComponents")
2  class TransformerEncoderBlock(tfkl.Layer):
3      def __init__(self, embed_dim, dense_dim, num_heads, dropout_rate=0.1, **
       kwargs):
4          # Initialise parent class
5          super().__init__(**kwargs)
6
7          # Store architecture parameters
8          self.embed_dim = embed_dim
9          self.dense_dim = dense_dim
10         self.num_heads = num_heads
11         self.dropout_rate = dropout_rate
12
```

```
        # Configure multi-head attention mechanism
        self.attention = tfkl.MultiHeadAttention(
            num_heads=num_heads,
            key_dim=embed_dim // num_heads,  # Scale dimension per attention head
            dropout=dropout_rate
        )

        # Configure feed-forward network
        self.dense_proj = keras.Sequential([
            tfkl.Dense(dense_dim, activation="relu"),
            tfkl.Dropout(dropout_rate),
            tfkl.Dense(embed_dim)
        ])

        # Configure normalisation layers
        self.layernorm_1 = tfkl.LayerNormalization(epsilon=1e-6)
        self.layernorm_2 = tfkl.LayerNormalization(epsilon=1e-6)

        # Configure dropout for residual connections
        self.dropout_1 = tfkl.Dropout(dropout_rate)
        self.dropout_2 = tfkl.Dropout(dropout_rate)

        self.supports_masking = True

    def call(self, inputs, training=False, mask=None):
        # Apply pre-norm layer normalisation
        normalized_inputs = self.layernorm_1(inputs)

        # Process padding mask for attention
        if mask is not None:
            padding_mask = tf.cast(mask[:, tf.newaxis, :], dtype="int32")
        else:
            padding_mask = None

        # Compute multi-head self-attention
        attention_output = self.attention(
            query=normalized_inputs,
            value=normalized_inputs,
            key=normalized_inputs,
            attention_mask=padding_mask,
            training=training
        )

        # Apply first residual connection
        attention_output = self.dropout_1(attention_output, training=training)
        first_residual = inputs + attention_output

        # Process through feed-forward network
        normalized_ffn = self.layernorm_2(first_residual)
        ffn_output = self.dense_proj(normalized_ffn)

        # Apply second residual connection
        ffn_output = self.dropout_2(ffn_output, training=training)
        return first_residual + ffn_output

```

```
    def get_config(self):
        # Enable layer serialisation
        config = super().get_config()
        config.update({
            "embed_dim": self.embed_dim,
            "dense_dim": self.dense_dim,
            "num_heads": self.num_heads,
            "dropout_rate": self.dropout_rate,
        })
        return config
```

Transformer Decoder Block

```
@keras.saving.register_keras_serializable(package="TransformerComponents")
class TransformerDecoderBlock(tfkl.Layer):
    def __init__(self, embed_dim, ff_dim, num_heads, dropout_rate=0.1, **
    kwargs):
        # Initialise parent class
        super().__init__(**kwargs)

        # Store architecture parameters
        self.embed_dim = embed_dim
        self.ff_dim = ff_dim  # Feed-forward network dimension
        self.num_heads = num_heads
        self.dropout_rate = dropout_rate

        # Configure masked self-attention mechanism
        self.self_attention = tfkl.MultiHeadAttention(
            num_heads=num_heads,
            key_dim=embed_dim // num_heads,  # Scaled dimension per head
            dropout=dropout_rate
        )

        # Configure cross-attention for encoder outputs
        self.cross_attention = tfkl.MultiHeadAttention(
            num_heads=num_heads,
            key_dim=embed_dim // num_heads,
            dropout=dropout_rate
        )

        # Configure feed-forward network
        self.ffn = keras.Sequential([
            tfkl.Dense(ff_dim, activation="relu"),
            tfkl.Dropout(dropout_rate),
            tfkl.Dense(embed_dim),
            tfkl.Dropout(dropout_rate)
        ])

        # Configure normalisation layers
        self.layernorm_1 = tfkl.LayerNormalization(epsilon=1e-6)
        self.layernorm_2 = tfkl.LayerNormalization(epsilon=1e-6)
        self.layernorm_3 = tfkl.LayerNormalization(epsilon=1e-6)

        # Configure dropout layers
        self.dropout_1 = tfkl.Dropout(dropout_rate)
        self.dropout_2 = tfkl.Dropout(dropout_rate)
        self.dropout_3 = tfkl.Dropout(dropout_rate)

```

```
        self.supports_masking = True

    def call(self, inputs, training=None):
        # Unpack inputs and encoder outputs
        inputs, encoder_outputs = inputs

        # Generate causal mask for autoregressive attention
        causal_mask = self.get_causal_attention_mask(inputs)

        # Apply masked self-attention
        attention_output_1 = self.self_attention(
            query=inputs,
            value=inputs,
            key=inputs,
            attention_mask=causal_mask,
            training=training
        )
        attention_output_1 = self.dropout_1(attention_output_1, training=
    training)
        out1 = self.layernorm_1(inputs + attention_output_1)

        # Apply cross-attention with encoder outputs
        attention_output_2 = self.cross_attention(
            query=out1,
            value=encoder_outputs,
            key=encoder_outputs,
            training=training
        )
        attention_output_2 = self.dropout_2(attention_output_2, training=
    training)
        out2 = self.layernorm_2(out1 + attention_output_2)

        # Process through feed-forward network
        ffn_output = self.ffn(out2, training=training)
        ffn_output = self.dropout_3(ffn_output, training=training)
        return self.layernorm_3(out2 + ffn_output)

    def get_causal_attention_mask(self, inputs):
        # Generate lower triangular mask for autoregressive attention
        input_shape = tf.shape(inputs)
        batch_size, sequence_length = input_shape[0], input_shape[1]
        i = tf.range(sequence_length)[:, None]
        j = tf.range(sequence_length)
        mask = tf.cast(i >= j, dtype="int32")
        mask = tf.reshape(mask, (1, sequence_length, sequence_length))
        mult = tf.concat(
            [tf.expand_dims(batch_size, -1), tf.constant([1, 1], dtype=tf.
    int32)],
            axis=0,
        )
        return tf.tile(mask, mult)

    def get_config(self):
        # Enable layer serialisation
        config = super().get_config()
        config.update({
            "embed_dim": self.embed_dim,
```

```
            "ff_dim": self.ff_dim,
            "num_heads": self.num_heads,
            "dropout_rate": self.dropout_rate
        })
        return config
```

28.4 Model Building

```
def build_transformer(
    vocab_size,
    sequence_length,
    embed_dim=128,
    latent_dim=1024,
    num_heads=4,
    num_encoder_layers=2,
    num_decoder_layers=2,
    name="transformer"
):
    # Configure encoder architecture
    encoder_inputs = keras.Input(shape=(None,), dtype="int64", name="
    encoder_inputs")
    x = PositionalEmbedding(sequence_length, vocab_size, embed_dim)(
    encoder_inputs)

    # Process through encoder blocks
    for _ in range(num_encoder_layers):
        x = TransformerEncoderBlock(embed_dim, latent_dim, num_heads)(x)
    encoder_outputs = x

    # Create encoder model
    encoder = keras.Model(encoder_inputs, encoder_outputs)

    # Configure decoder architecture
    decoder_inputs = keras.Input(shape=(None,), dtype="int64", name="
    decoder_inputs")
    encoded_seq_inputs = keras.Input(shape=(None, embed_dim), name="
    decoder_state_inputs")

    # Apply positional embedding to decoder inputs
    x = PositionalEmbedding(sequence_length, vocab_size, embed_dim)(
    decoder_inputs)

    # Process through decoder blocks
    for _ in range(num_decoder_layers):
        x = TransformerDecoderBlock(embed_dim, latent_dim, num_heads)([x,
    encoder_outputs])

    # Generate output probabilities
    decoder_outputs = tfkl.Dense(vocab_size, activation="softmax")(x)
    decoder = keras.Model([decoder_inputs, encoded_seq_inputs],
    decoder_outputs)

    # Construct complete transformer model
    transformer = keras.Model(
        {
            "encoder_inputs": encoder_inputs,
```

```
            "decoder_inputs": decoder_inputs
        },
        decoder_outputs,
        name=name
    )

    # Print model architecture summary
    print(f"Transformer architecture:\n"
          f"  Vocabulary size:    {vocab_size:>8,d}\n"
          f"  Sequence length:    {sequence_length:>8d}\n"
          f"  Embedding dim:      {embed_dim:>8d}\n"
          f"  Latent dim:         {latent_dim:>8d}\n"
          f"  Attention heads:    {num_heads:>8d}\n"
          f"  Encoder layers:     {num_encoder_layers:>8d}\n"
          f"  Decoder layers:     {num_decoder_layers:>8d}")

    return transformer, encoder, decoder
```

28.5 Compile and Train

```
transformer, encoder, decoder = build_transformer(
    vocab_size=vocab_size,
    sequence_length=sequence_length,
    embed_dim=embed_dim,
    latent_dim=latent_dim,
    num_heads=num_heads,
    num_encoder_layers=num_encoder_layers,
    num_decoder_layers=num_decoder_layers
)

# Display model information
transformer.summary(expand_nested=True, show_trainable=True)

# Create visual representation
tf.keras.utils.plot_model(
    transformer,
    show_trainable=True,
    expand_nested=True,
    dpi=70,
    show_shapes=True
)
```

```
# Initialise Adam optimiser with default parameters
optimizer = tfk.optimizers.Adam()

# Configure model with loss function and metrics
transformer.compile(
    optimizer=optimizer,
    loss=tfk.losses.SparseCategoricalCrossentropy(ignore_class=0),  # IMPORTANT: Ignore padding tokens
    metrics=['accuracy']
)

# Print training configuration
print(f"Training configuration:\n"
      f"  Optimiser:          Adam\n"
```

```
      f"  Learning rate:      {optimizer.learning_rate.numpy():>8.2e}\n"
      f"  Loss function:      Sparse Categorical Cross-entropy\n"
      f"  Metrics:            Accuracy")
```

```
# Configure training callbacks
callbacks = [
    # Monitor validation accuracy for early stopping
    tfk.callbacks.EarlyStopping(
        monitor='val_accuracy',
        mode='max',
        patience=5,
        restore_best_weights=True,
        verbose=1
    ),
    # Adjust learning rate based on performance
    tfk.callbacks.ReduceLROnPlateau(
        monitor='val_accuracy',
        factor=0.1,
        patience=3,
        min_lr=1e-6,
        mode='max',
        verbose=1
    )
]

# Print training parameters
print(f"Training configuration:\n"
      f"  Maximum epochs:     {100:>8d}\n"
      f"  Early stop patience:{5:>8d}\n"
      f"  LR reduce patience: {3:>8d}\n"
      f"  Minimum LR:         {1e-6:>8.0e}")

# Execute training process
history = transformer.fit(
    train_ds,
    validation_data=val_ds,
    epochs=100,
    callbacks=callbacks
).history

# Evaluate model performance
final_val_accuracy = round(max(history['val_accuracy']) * 100, 2)
print(f"\nTraining results:\n"
      f"  Final validation accuracy: {final_val_accuracy:>8.2f}%\n"
      f"  Total epochs trained:      {len(history['accuracy']):>8d}")

# Save trained model
model_filename = f'transformer_seq2seq_{final_val_accuracy}.keras'
transformer.save(model_filename)
print(f"  Model saved as:            {model_filename}")

# Release model resources
del transformer
```

References

Boracchi, G. (2024). *Artificial neural networks and deep learning.* Retrieved from https://drive.google.com/drive/folders/1JMiRhyfP5Aqdr8qBtb9CMaIkuwWd62CR?usp=drive_link (Accessed: 2025-02-15)

Lomurno, E. (2024). *[2024-2025] an2dl.* Retrieved from ttpsh://boracchi.faculty.polimi.it/teaching/AN2DL.htm (Accessed: 2025-02-15)

Matteucci, M. (2024). *Artificial neural networks and deep learning.* Retrieved from https://chrome.deib.polimi.it/index.php?title=Artificial_Neural_Networks_and_Deep_Learning (Accessed: 2025-02-15)

Morbidelli, P., Carrera, D., Rossi, B., Fragneto, P., & Boracchi, G. (2020). Augmented grad-cam: Heat-maps super resolution through augmentation. In *Icassp 2020 - 2020 ieee international conference on acoustics, speech and signal processing (icassp)* (p. 4067-4071). doi: 10.1109/ICASSP40776.2020.9054416

Petsiuk, V., Das, A., & Saenko, K. (2018). *Rise: Randomized input sampling for explanation of black-box models.* Retrieved from https://arxiv.org/abs/1806.07421

Wikipedia contributors. (2024a). *Disjunctive normal form.* Retrieved from https://en.wikipedia.org/wiki/Disjunctive_normal_form (Accessed: 2025-10-15)

Wikipedia contributors. (2024b). *One-hot.* Retrieved from https://en.wikipedia.org/wiki/One-hot#Machine_learning_and_statistics (Accessed: 2024-10-17)