Technische Universität München

Summary of the lecture I2DL

Introduction to Deep Learning

(IN2346)

based on the lectures of
Prof. Niessner, Prof. Leal-Taixé, Prof. Dai, and Prof. Cremers

Authors:

Dan Halperin, Benjamin Heltzel

Compiled on December 12, 2024 at 05:57:33

Contents

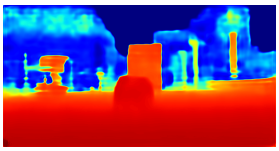
I Machine Learning Basics

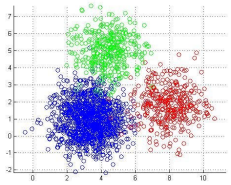
I.1 Tasks in Machine Learning

- Supervised learning: supervised learning is a type of learning, where the model is trained on an existing and **fixed** ground truth, relative to the task at hand: Classes for classification, depth values for depth estimation, facial key-points or the next correct letter in the sentence.
- Unsupervised learning: unsupervised learning is a type of learning, where the model is trained on data without any ground truth. The model is supposed to find patterns in the data, which can be used for clustering, dimensionality reduction or generative tasks (E.g. Autoencoders).
- Classification: a discrete goal, where the model assigns the input to one of the C classes. Also applies to semantic segmentation of images, where each pixel is assigned to one of the C classes. This can be done in a supervised way (given labeled data) or unsupervised way (clustering). Examples: image classification, object detection, semantic segmentation, sentence completion, etc.
- Regression: a **continuous** goal, where the model predicts a continuous value, either bound or unbound to some range. Examples are depth estimation, facial key-point detection, etc.


Original


Segmentation


Depth Prediction


Clustering

I.2 Datasets and Dataloaders

NOTE: This section refers to Exercise 03: Datasets and Data loaders.

I.2.a Datasets - general notes

- Datasets are the core of machine learning. They are the **most** important part, and their quality and quantity can make or break a project. Machine learning models are **data-driven**, meaning that all they can learn and use is within the data they are given.
- Datasets are usually split into three parts: training, validation and test set.
  - First, the **training set** is used to train the model. It is the only **seen** data that the model is exposed to, and always dealt the bigger chunk of the overall dataset.
  - The **validation set** is an unseen part of the dataset. It is commonly stated that it is used for tuning the hyperparameters of the model - but that it is only before we even start training. More importantly, it is used to **evaluate** the model during training. This is important, because the

model can overfit (subsection IV.2.a) the training data, and the validation set is the main-used tool to detect this.

  - The **test set** is the final part of the dataset, and should be kept biased as little as possible (no optimization choice was taken according to it). It is common to argue that this split should be used **only once** - however, that is only because it only represents a truly unseen data, if we, as developers cannot make architectural choices based on the performance of the test set.
  - It is important to shuffle the dataset, otherwise different splits could be very biased, e.g. - the training set would contain only images that were taken during day-time, and the test would consist only of nighttime images.
  - Common split ratios are {80%, 10%, 10%}, {70%, 15%, 15%}, etc. The training set is always the biggest, as it is the only data that the model is trained on.

- It is important to shuffle the dataset, otherwise different splits could be very biased, e.g. - the training set would contain only images that were taken during day-time, and the test would consist only of nighttime images.
- A dataset represents a distribution, which is a sub-part of the real world distribution. It is limited by the available data, and no matter how big it is, a modern finite dataset cannot represent the real world distribution.

I.2.b Datasets - Programming notes

Usually, in the modern deep learning frameworks, the datasets are represented as classes. This class has two main methods that need to be implemented:

- `__len__` - this method returns the length of the dataset. It is used by the data loader to know how many samples are in the dataset.
- `__getitem__` - this method returns the i-th element of the dataset. Note, that it returns a single sample, and NOT a batch of samples. The dataloader is responsible for calling this function multiple times to get a batch of samples. Within this function, the sample can be loaded from the disk, normalized, transformed, changed, etc, alongside with the label, if exists.

I.2.c Data augmentation

Data augmentation is a technique used to artificially increase the size of the dataset. It is especially useful when the dataset is small, and the model is prone to overfitting. It is a common practice to use data augmentation in computer vision tasks, where the images can be flipped, rotated, cropped, etc.

- Is considered a **Regularization** technique (subsection IV.2.d) as it helps the model to generalize better.
- It does NOT increase the physical size of the dataset, but at each epoch, the model sees a different version of the data with some probability p.
- Unlike normalization and other preprocessing, data augmentation is applied only to the training set, and not to the validation and test sets.
- While the idea is good and a model could really benefit from data augmentation, the techniques should fit the problem at hand. For example, it is not a good idea to flip the images in a medical imaging task, as the left and the right side of the body are different, or rotation by 180° of images of numbers, as the number 6 would become 9, etc.

I.2.d Data processing

Data processing is an overall name for techniques that are applied to the data before it is fed to the model:

- Normalization: this is a technique used to scale the data to a certain range. It is crucial that the input would have a reasonable magnitude of value, so the gradients wouldn't explode or vanish (subsection IV.2.b). Also, it makes the input to the network more consistent and eases the learning process. It is a common practice to normalize the data to the range $[0, 1]$ or $[-1, 1]$. This is done by calculation the mean and the standard deviation of the training set, and then normalizing the data by subtracting the mean and dividing by the standard deviation:

$$x_{\text{norm}} = \frac{x - \mu}{\sigma}$$

- Those techniques are applied to each sample of the dataset, and are usually done within the `__getitem__` method of the dataset class. Unlike data augmentation, they are applied to all the splits of the dataset, as the model expects the data to be in the same format.

I.2.e Dataloaders

Dataloaders are classes that are used to load the data from the dataset, and to create batches of samples. They are responsible for shuffling the data, creating batches, They are usually used in the training loop, where the model is trained on the data.

I.3 Exam questions

I.3.a Questions

1. What is the difference between supervised and unsupervised learning?
2. What is the difference between classification and regression?
3. What advantages does unsupervised learning have over supervised learning?
4. Give example from the class where we utilized unsupervised learning to improve the accuracy.
5. K-means:
   a) What is the K-mean clustering algorithm?
   b) What is its key hyperparameter?
   c) What are the consequences of a too small or too large value of its hyperparameter?
6. PCA:
   a) What is PCA?
   b) What Deep Learning architecture could perform a similar task? Why is the deep learning method preferable for real world problems?
7. Data splits:
   a) Why is it important to shuffle the data before splitting it??
   b) What are common split rations?
   c) Why is the training set bigger?

d) When can we relax the ratio between the splits to be more even? When the other way around?

e) What would be a consequence of a too small validation set?

f) What can we do as simple deep learning engineers to overcome a too small training set?

8. Why not to use the validation set also within the training set and just use the test for validation?
9. What is the main issue with modern datasets' sizes? Why is it so?
10. To which part of the dataset should we apply data augmentation? Why?
11. To that regard, what is the difference between data augmentation and data processing techniques like normalization?
12. Why is the mean and std of the data calculated only over the training set?

I.3.b Answers

1. Supervised learning is a type of learning, where the model is trained on an existing and **fixed** ground truth, relative to the task at hand: Classes for classification, depth values for depth estimation, facial key-points or the next correct letter in the sentence. Unsupervised learning is a type of learning, where the model is trained on data without any ground truth. The model is supposed to find patterns in the data, which can be used for clustering, dimensionality reduction or generative tasks.
2. Classification is a discrete goal, where the model assigns the input to one of the C classes. Regression is a continuous goal, where the model predicts a continuous value, either bound or unbound to some range. Examples are depth estimation, facial key-point detection, etc.
3. Labeled data is expensive, and sometimes impossible to get. Unsupervised learning can be used to find patterns in the data, that can be used for other tasks, like classification or regression, without the need to manually or artificially label the data, usually with human-made errors. Note (irrelevant for the exam): It was shown that unsupervised settings allow a much richer representation learned features, for example: when GPT-1 was trained, it was shown that the model learned to detect the sentiment of the text (good or bad), even though it was trained on a next-letter completion task.
4. Autoencoder. We used an autoencoder to reconstruct unlabeled data, to create a feature extractor, and then used this encoder to train a classifier on the much smaller labeled dataset (Exercise 08).
5. a) K-mean clustering is an unsupervised learning algorithm, that is used to cluster the data into K clusters. It is an iterative algorithm, where the data is assigned to the closest cluster, and then the cluster's center is recalculated. This is done until the algorithm converges.

   b) The key hyperparameter of the K-mean clustering algorithm is the number of clusters K.

   c) If the value of K is too small, the algorithm could merge clusters that are not similar (underfitting) and if the value of K is too large, the algorithm could split clusters that are similar (overfitting).

6. a) PCA is a **linear** dimensionality reduction technique, that is used to reduce the number of features of the data, while keeping the most important features.

   b) A deep learning architecture that could perform a similar task is an autoencoder. The autoencoder is a **non-linear** dimensionality reduction technique. The deep learning method is preferable for real world problems, as it can learn non-linear features, and can be used for more complex tasks, like classification, regression, etc.

7. a) It is important to shuffle the dataset, as if the data is not shuffled it could lead to biased splits, that do not represent the same overall distribution. Example: if the datasets consist of day-time and nighttime images sorted accordingly, the training set might hold only day-time images, while

the test would hold only the much more difficult nighttime ones. Also, the model could learn the order of the data, and not the data itself.

b) Common split ratios are {80%, 10%, 10%}, {70%, 15%, 15%}, etc.

c) The training set is always the largest, as it is the only data that the model is trained on.

d) We can relax the ratio between the splits to be more even, when the dataset is big enough, and the model is not prone to overfitting.

e) A consequence of a too small validation set could be that the model could overfit the training set, and the validation set would not be able to detect this.

f) To overcome a too small training set, we can use data augmentation, as it artificially increases the size of the dataset (not the physical size, but the number of samples that the model sees).

8. With no validation set, we cannot detect overfitting problems. On the other hand, the test set has a very clear purpose of being the most unbiased unseen data segment of the dataset, and should not be used to validate our training process.

9. The main issue with modern datasets' sizes is that they are finite and limited by the available data, so they cannot represent the real world distribution. Data collection is very expansive in terms of both time and resources.

10. Data augmentation is applied only to the training set, as a mean of Regularization, to help the model generalize better and to prevent overfitting. It is not applied to the validation and test sets, as they are supposed to represent the original problem's data distribution, and the model should be evaluated on that distribution. If applied, the performance on those datasets would be worse and not comparable to other models.

11. The difference between data augmentation and data processing techniques like normalization is that data augmentation is applied to the training set, and is used to artificially increase the size of the dataset, while data processing techniques are applied to all the splits of the dataset, and are used to preprocess the data before it is fed to the model.

12. The mean and std of the data is calculated only over the training set, as we must keep the test set as unbiased as possible. That is, no information from the test set should be used to train the model, and the mean and std of the data are part of the information that the model could learn.

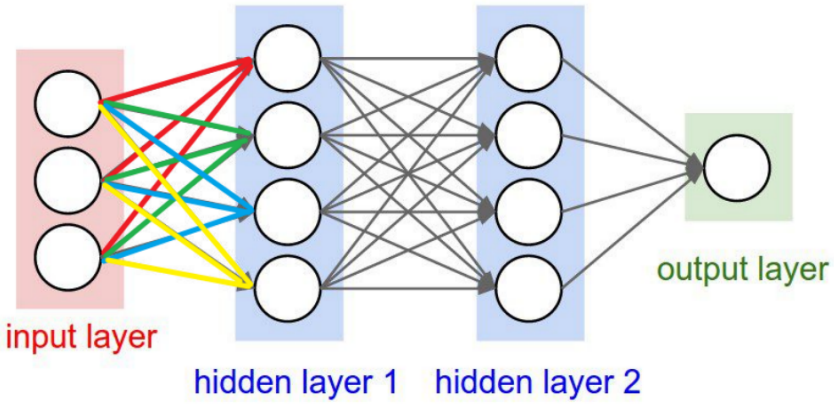
II Neural Networks

II.1 Neural Networks

Neural networks are simply defined as a series of functions $f(x), g(x), h(x), \ldots$ that are composed together:

$$y = h(g(f(x)))$$

A neural network could be described in the following sketch:



In the drawing, the circles (nodes) represent the **neurons**, and the lines (edges) represent the connections between the neurons (**the weights**). Each neuron holds **a single value**. A common way to index those neurons would be $x_{l,i,o}$, where l is the layer, i is the index of the neuron in the input layer, and o is the index of the neuron in the output layer. Usually, neural networks represent **non-linear** functions. For that, we apply a non-linear **activation function** to the output of each neuron of a hidden layer, but it is not shown in this drawing.

On top of that, each set of lines between any two groups of neurons is what we refer to as the **layers**, or the functions f, g, h that are stated above. The first layer is called the **input layer**, the last layer is called the **output layer**, and the layers in between are called the **hidden layers**. The number of hidden layers and the number of neurons in each layer are called the **architecture** of the neural network. The architecture of the neural network is a hyperparameter that we need to tune to get the best performance.

Another crucial point, is that in the sketch above we see a single input flowing through the network, and not a batch of them. However, in practice with computers, we manage to feed into the networks a batch of inputs at once, all computed in parallel, but independently.

In matrix notation, let's assume the affine layer, or in its common name: **fully connected layer**, is defined as:

$$y = XW + b$$

Those variables are usually represented as matrices:

$$X \in \mathbb{R}^{N \times D}, W \in \mathbb{R}^{D \times M}, b \in \mathbb{R}^{M}$$

where N is the number of samples in the batch, D is the number of features in each sample, and M is the number of neurons in the layer. The output y is of shape $\mathbb{R}^{N\times M}$.

For the first layer in the example above, we assume that

$$X \in \mathbb{R}^{N\times 3}, W \in \mathbb{R}^{3\times 4}, b \in \mathbb{R}^4$$

or in matrices:

$$X = \begin{bmatrix} x_{1,1} & x_{1,2} & x_{1,3} \\ x_{2,1} & x_{2,2} & x_{2,3} \\ \vdots & \vdots & \vdots \\ x_{N,1} & x_{N,2} & x_{N,3} \end{bmatrix}, W = \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} & w_{1,4} \\ w_{2,1} & w_{2,2} & w_{2,3} & w_{2,4} \\ w_{3,1} & w_{3,2} & w_{3,3} & w_{3,4} \end{bmatrix}, b = \begin{bmatrix} b_1 & b_2 & b_3 & b_4 \end{bmatrix}$$

That means that each b_i in b corresponds to one feature in a row of XW - but adding them like that, it is quite impossible mathematically:

- NumPy or PyTorch use **broadcasting** to duplicate the row vector b to be a matrix B_{NxM}, by simply copying b N times and stacking them together along the rows, or the 0-axis. But that's just the programming application.
- Mathematically speaking, it's not $Y = XW + b$, but $Y = XW + 1^N b$, where 1^N is a column vector, such that

  $$\begin{bmatrix} 1_1 \\ \vdots \\ 1_N \end{bmatrix} \begin{bmatrix} b_1 & \dots & b_M \end{bmatrix} = \begin{bmatrix} b_1 & \dots & b_M \\ \vdots & \ddots & \vdots \\ b_1 & \dots & b_M \end{bmatrix}_{N\times M}$$

  That gives us the broadcast that python does by itself, and allows us to actually sum those matrices together.

II.1.a Fully Connected Layer

The fully connected layer (FC) is a very common computational layer in neural networks. It is also known as the **affine layer**, or the **dense layer**. The fully connected layer is defined as:

$$y = XW + b$$

where $X_{N\times D}$ is the input data, $W_{D\times M}$ is the weight matrix, and $b_{1\times M}$ is the bias vector. Both W and b are learnable variables. Please check out the sketch above for a visual representation. In addition, a neural network that is based on FC layers as its main building blocks is called a Fully Connected network (FCN). You could also find it under the name of a **Multi-Layer Perceptron** (MLP).

Important to note, that each neuron in the input layer is connected to each neuron in the output layer. This means that the number of weights in the fully connected layer is equal to the number of neurons in the input layer times the number of neurons in the output layer. The number of biases in the fully connected layer is equal to the number of neurons in the output layer. Therefore, we claim that FC layers capture **global** features - but would struggle with **local** ones. To that end, we use **convolutional layers**, as described later on (chapter III).

II.1.b Backpropagation

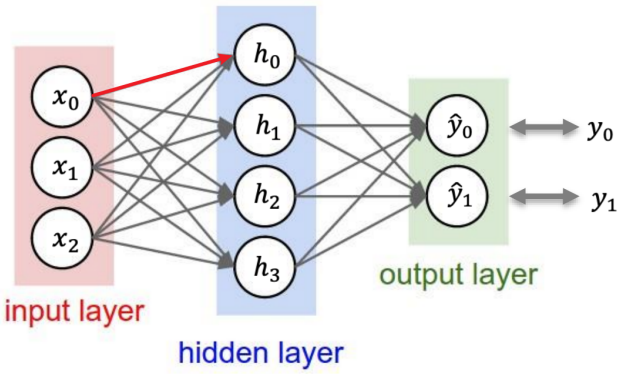
A pro tip: the nominator is ALWAYS a function, and the denominator is always the input to that function. For example, given a function $y = ReLU(xw + b)$, it's better to write:

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y}\frac{\partial ReLU(z)}{\partial z}\frac{\partial (xw+b)}{\partial x}$$

than

$$\frac{\partial L}{\partial y}\frac{\partial y}{\partial z}\frac{\partial z}{\partial x}$$

because it's easier on our brains. Substitute the value with the actual function for visual convenience, at least when dealing with **matrix notation**.



Assume the notation, $w_{l,i,o}$, where l, i and o are the layer, input neuron and output neuron indices. Let us find the derivative of the weight $w_{1,1,1}$ colored in red, according to some loss value $L \rightarrow \frac{\partial L}{\partial w_{1,1,1}}$:

Forward pass:

- $h = XW_1 + b_1$
- $\hat{y} = hW_2 + b_2$
- $L = Loss(\hat{y})$
- For simplicity, assume no activation.

Backward pass:

- $\frac{\partial L}{\partial w_{1,1,1}} = \frac{\partial L}{\partial \hat{y}}\frac{\partial (hW_2+b_2)}{\partial h}\frac{\partial (XW_1+b_1)}{\partial w_{1,1,1}}$
- $\frac{\partial L}{\partial w_{1,1,1}} = \sum_{i=1}^{4}\frac{\partial L}{\partial h_i}\frac{\partial h_i}{\partial w_{1,1,1}}$
- $\frac{\partial L}{\partial h_i} = \sum_{j=1}^{2}\frac{\partial L}{\partial \hat{y}_j}\frac{\partial \hat{y}_j}{\partial h_i}$
- $\frac{\partial L}{\partial w_{1,1,1}} = \sum_{j=1}^{2}\sum_{i=1}^{4}\frac{\partial L}{\partial \hat{y}_j}\frac{\partial \hat{y}_j}{\partial h_i}\frac{\partial h_i}{\partial w_{1,1,1}}$
- Note that here I found it more convenient to remain with the variable names, as we're dealing with matrix entries.

In matrix notation, the **gradients** of the affine layer's variables in respect to the loss function (a scalar) are:

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial y}\cdot W^{\intercal}$$

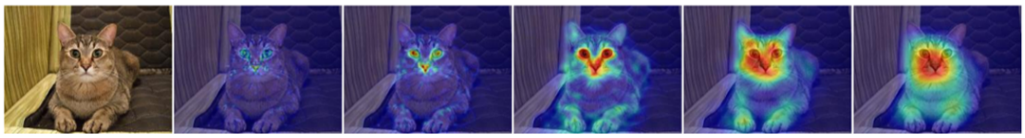
$$\frac{\partial L}{\partial W} = X^\intercal \cdot \frac{\partial L}{\partial y}$$
$$\frac{\partial L}{\partial b} = \begin{bmatrix} 1 & 1 & \dots & 1 \end{bmatrix}^N \cdot \frac{\partial L}{\partial y}$$

Where $L \in \mathbb{R}$ is the loss value, and $y \in \mathbb{R}^{N \times M}$ is the output of the layer. For more details about how to derive them, please refer to chapter VII.

II.2 Activation functions

The main computing power of a neural network are the **linear** layers, that contain the learnable weights of the model. However, linear functions are not sufficient to learn complex patterns in the data. Therefore, we introduce **non-linear** functions, called **activation functions**, that are applied to the output of the linear layers. These layers are what make neural networks **universal function approximations**, and capable of learning patterns and relevant features in real world data. In the following section, we will cover the most common activation functions used in neural networks, and discuss their pros and cons.

II.2.a Terms



- Activation map / Feature map - an output of a linear layer, after applying an activation function. A tensor of numbers that represents some learnt features from the previous layer.
- Features - what a group of individual numbers represent together. A single neuron holding a value has no meaning but within the context of its neighbors or all other neurons in the layer, depending on the linear layer it is created with (Convolutions vs. Affine Layers).
- Zero centered - a property of an activation function, that the output of the function could be both positive and negative. The mean doesn't have to be actually 0 (E.g. LeakyReLU). When an activation is not zero-centered, it could introduce bias in the network, and could lead to less optimal step direction (All gradients are positive or negative). However - for the "zigzag" behavior to actually happen, it boils down to the loss function used.

  Example: Assume the following network -

  - $Y_1 = XW_1 + b_1$
  - $Z_1 = \sigma(Y_1)$
  - $Y_2 = Z_1W_2 + b_2$
  - $Z_2 = \sigma(Y_2)$
  - $L(Z_2) = l \in \mathbb{R}$

  In the plot above one could see that the MAE ($|x|$) and MSE (x^2) both have negative and positive slopes, while the negative logarithm (CE / BCE), at the effective range of [0, 1], has only a negative slope. On the other hand, Z_1 is all non-negative by the definition of the sigmoid function. Therefore,

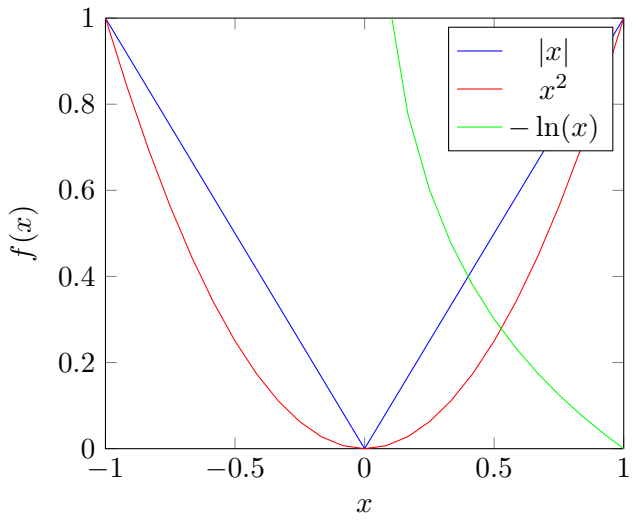


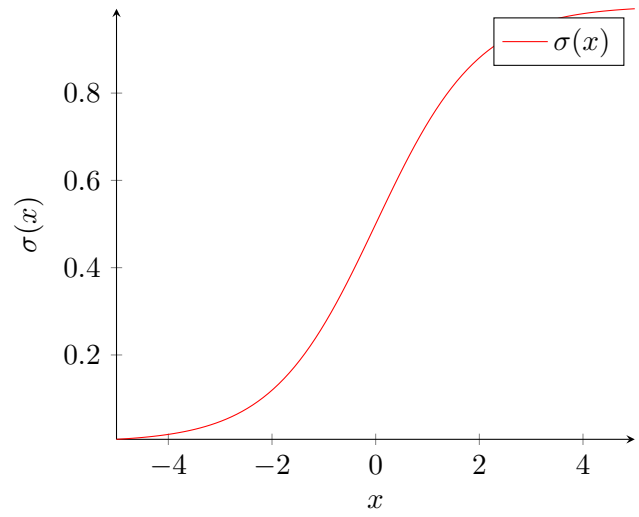
Figure II.1: Graphs of |x|, x^2 and $-log(x)$ in the range of -1 to 1

if using the cross-entropy loss functions, the gradient:

$$\frac{\partial l}{\partial W_2} = Z_1^\top \cdot (Z_2(1 - Z_2) * \frac{\partial l}{\partial Z_2})$$

is always non-positive, and the weights are always updated in the same direction.

II.2.b Sigmoid



The sigmoid function is defined as:

$$\mathbb{R} \to [0, 1]$$
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Main features:

- It is a scalar function, that operates only on scalars. Therefore, it is applied in an **element-wise** fashion to the tensor-like input (on each entry independently).

- It projects the input to the range $[0, 1]$, so it can be interpreted as the probability of the input belonging to a class.
- It is differentiable, and has a simple derivative:

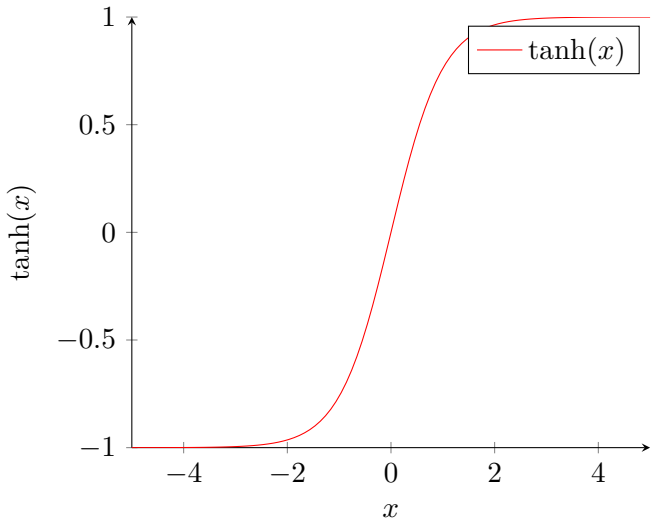
$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1 - \sigma(x))$$

  Note that this derivative is independent of the input x - this variable is not needed for the calculation of the derivative.
- Usually used in binary-classification, to make the network's logits probability-like, to be used in the binary-cross-entropy loss function.

Drawbacks:

- The sigmoid function saturates when the input is very large or very small. Its effective range is roughly [-3.5, 3.5]. Beyond those boundaries, the gradient of the function is very close to zero - **saturates**.
- Largest gradient is at $x = 0 \rightarrow \sigma(0) = 0.5 \rightarrow \sigma'(0) = \sigma(0)(1 - \sigma(0)) = 0.25$. This means that the gradient is very small, which has a negative impact on the learning process.
- It is not zero-centered.

II.2.c Hyperbolic Tangent - Tanh



The hyperbolic tangent function is defined as:

$$\mathbb{R} \rightarrow [-1, 1]$$

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Main features:

- Also a scalar function. Therefore, it is applied in an **element-wise** fashion to the tensor-like input.
- It projects the input to the range $[-1, 1]$, and is zero-centered.
- It is differentiable, and has a simple derivative:

$$\frac{\partial \tanh(x)}{\partial x} = 1 - \tanh^2(x)$$

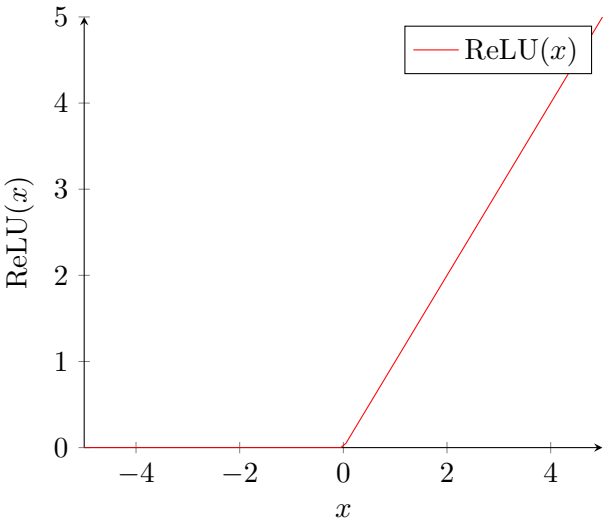
- The highest gradient is at $x = 0 \rightarrow \tanh(0) = 0 \rightarrow \tanh'(0) = 1$, which is already an improvement over the sigmoid function, and has much more lively gradients within its effective range.
- It could be seen as a scaled version of the sigmoid function:

$$\tanh(x) = 2\sigma(2x) - 1$$

Drawbacks:

- The Tanh function saturates when the input is very large or very small. It's effective range is roughly [-2, 2], which is very limiting. Beyond those boundaries, the gradient of the function is very close to zero - **saturates**.

II.2.d Rectified Linear Unit - ReLU



The rectified linear unit function is defined as:

$$\mathbb{R} \rightarrow [0, \infty)$$

$$\text{ReLU}(x) = \max(0, x)$$

Main features:

- It is a scalar function, that operates only on scalars. Therefore, it is applied in an **element-wise** fashion to the tensor-like input.
- It is a simple and computationally efficient function.
- It is non-linear, and has a simple derivative:

$$\frac{\partial \text{ReLU}(x)}{\partial x} = \begin{cases} 0 & \text{if } x \leq 0 \\ 1 & \text{if } x > 0 \end{cases}$$

  However, it is not continuous and not differentiable at $x = 0$.
- While it saturates for negative values, and so does its gradient in the range of $(-\infty, 0)$, it has a full-magnitude gradient for positive values, and is not limited in its effective range.

Drawbacks:

- The ReLU function saturates when the input is negative. This means that the gradient of the function is zero for negative values, and the weights leading to those neurons are not updated during backpropagation. This is known as the **dying ReLU** problem.
- It is not zero-centered.

Modern alternatives: GELU, ELU, LeakyReLU, etc.

II.2.e The "vanishing gradient" vs the "Dying ReLU" problems

These two terms often repeat in the literature, and are usually confused. Even modern language models (as for 2024), repeat the hesitant and incomplete definition they learned from on the internet.

1. **The vanishing gradient problem**: As mentioned above, some activation functions produce very small gradients for certain values of the input. For very small networks, this doesn't pose a problem. However, when considering the architecture of much deeper networks, then we could observe a very negative behavior. Since the gradients of those activations are small to begin with, when multiplied by such small gradients of layers "up-the-stream", they shrink exponentially. That causes a phenomenon, where the weights of the **shallower** layers (closer to the input) are not updated as much as the weights of the **deeper** layers (closer to the output), if at all - they could simply "vanish".

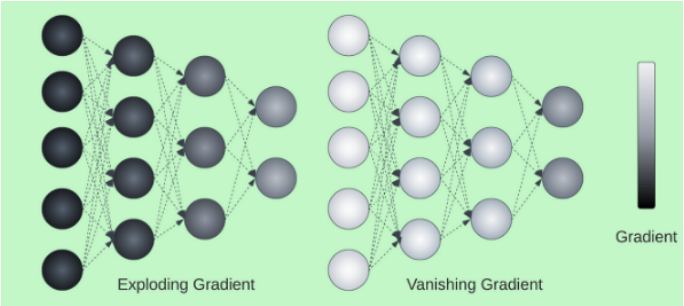


Figure II.2: In this example, we could see how the magnitude of the gradients is exponentially decreasing as they flow "up the stream", from the loss function to the input layer, during backpropogation.

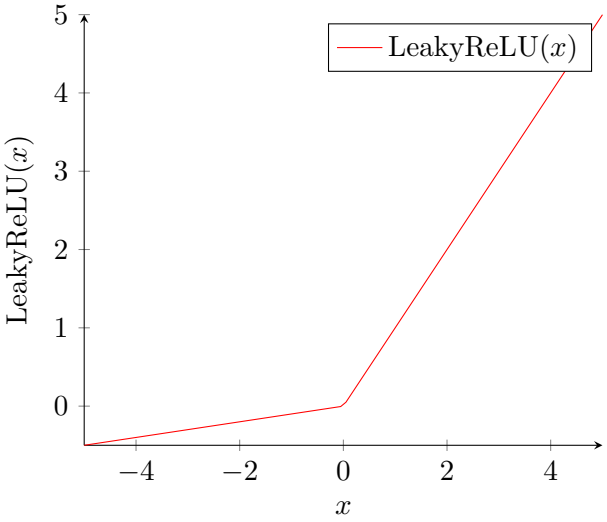
2. **The dying ReLU problem**: The ReLU function saturates when the input is negative. This means that the gradient of the function is zero for negative values, and the weights leading to those neurons are not updated during backpropagation. The neurons that are "dead" are not activated, and do not contribute to the learning process. This can cause the network to learn suboptimal features, and can slow down the learning process. However, it is true in the scope of a single iteration of the training process, and doesn't mean the weights will not be updated at some stage. However, in some very rare cases - a majority of the neurons could be "dead", and the network would not learn at all.
3. The difference between the two: The vanishing gradient problem is a gradual problem, the usually occurs in very deep networks, given certain activation functions. On the other hand, the dying ReLU is an immediate problem, that influence the training process, or the performance differently. Both could happen at the same time, or not at all.

How should we then deal with those problems?

- **The vanishing gradient problem**:
  - Use activation functions that have a full-magnitude gradient in their effective range, such as the ReLU function or its variants.

  - Use normalization techniques such as batch normalization or layer normalization to stabilize the gradients during training.
  - Use skip connections or residual connections to allow the gradients to flow more easily through the network, with the "highway" of gradients.
  - Use gradient clipping to prevent the gradients from becoming too large or too small.
  - Use initialization techniques, such as the Xavier initialization, to prevent the neurons from saturating (subsubsection IV.2.g).
- **The dying ReLU problem**:
  - Use the more updated variants of ReLU, such as the Leaky ReLU, Parametric ReLU, or Exponential Linear Unit (ELU) functions.
  - Use initialization techniques such as the Kaiming He initialization, that is tailor-made for ReLU, to prevent the neurons from saturating

II.2.f Leaky ReLU



The leaky rectified linear unit function is defined as:

$$\mathbb{R} \to \mathbb{R}$$

$$\text{LeakyReLU}(x) = \max(\alpha x, x)$$

Main features:

- It is a scalar function, operates only on scalars. Therefore, it is applied in an **element-wise** fashion to the tensor-like input.
- The variable $\alpha \in \mathbb{R}$ can take any value, but remains **fixed** after it is set. Originally set to 0.1.
- It is a simple and computationally efficient function, and is a variant of the ReLU function. However, costs a bit more computationally than simple ReLU.
- It is non-linear, and has a simple derivative:

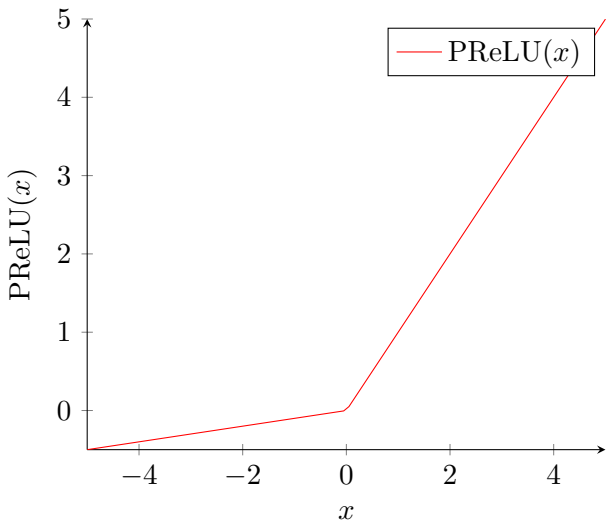
$$\frac{\partial \text{LeakyReLU}(x)}{\partial x} = \begin{cases} \alpha & \text{if } x \leq 0 \\ 1 & \text{if } x > 0 \end{cases}$$

- It is a "zero centered" function.
- Solves the "dying ReLU" problem.
- The hyperparameter α is usually set to a small value, such as 0.01. However - it is not defined anywhere that it must be positive, and therefore, one could also set it to a negative value.

Drawbacks:

- The Leaky ReLU function is not differentiable at $x = 0$.
- Introduces hyperparameter, which needs to be tuned.
- A bit more computationally expensive than the ReLU function.

II.2.g Parametric ReLU



The parametric rectified linear unit function is defined as:

$$\mathbb{R} \to \mathbb{R}$$

$$\text{PReLU}(x) = \max(\alpha x, x)$$

Main features:

- It is a scalar function, operates only on scalars. Therefore, it is applied in an **element-wise** fashion the tensor-like input.
- The variable $\alpha \in \mathbb{R}$ is learnable, and changes during training through backpropagation. This allows the network to learn the optimal value of α for each layer.
- It is non-linear, and has a simple derivative:

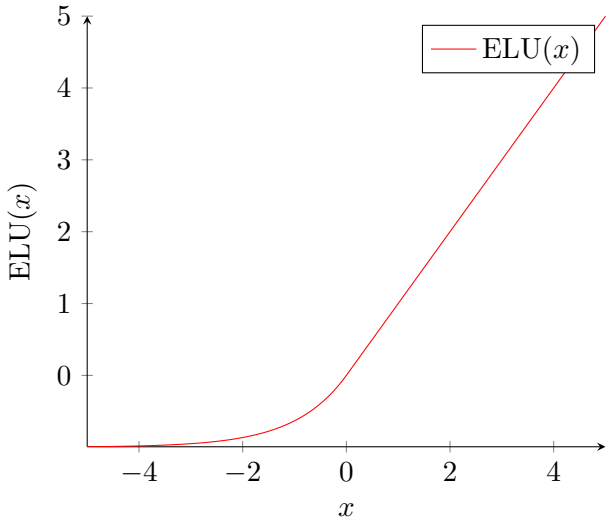
$$\frac{\partial \text{PReLU}(x)}{\partial x} = \begin{cases} \alpha & \text{if } x \leq 0 \\ 1 & \text{if } x > 0 \end{cases}$$

- It could be a zero-centered function, depending on the learnt α.
- Solves the "dying ReLU" problem.

Drawbacks:

- The Parametric ReLU function is not differentiable at $x = 0$.
- A bit more computationally expensive than the ReLU function.

II.2.h Exponential Linear Unit - ELU



The exponential linear unit function is defined as:

$$\mathbb{R} \to \mathbb{R}$$

$$\text{ELU}(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha(e^x - 1) & \text{if } x < 0 \end{cases}$$

Main features:

- It is a scalar function, operates only on scalars. Therefore, it is applied in an **element-wise** fashion to the tensor-like input.
- It is a simple and computationally efficient function, and is a variant of the ReLU function. However, costs a bit more computationally.
- It is non-linear, and has a simple derivative:

$$\frac{\partial \text{ELU}(x)}{\partial x} = \begin{cases} 1 & \text{if } x \geq 0 \\ \alpha e^x & \text{if } x < 0 \end{cases}$$

- It is a "zero centered" function.
- Solves the "dying ReLU" problem.

Drawbacks:

- The ELU function is not differentiable at $x = 0$ for any $\alpha \neq 1$ and can cause numerical instability during training. However, in practice, this is not a major issue.
- Introduces hyperparameter, which needs to be tuned.
- A bit more computationally expensive than the ReLU function.

II.2.i MaxOut

The MaxOut function is defined as:

$$\mathbb{R} \to \mathbb{R}$$

$$\text{MaxOut}(x) = \max(w_1^T x + b_1, w_2^T x + b_2)$$

It is non-linear, and has a simple derivative:

$$\frac{\partial \text{MaxOut}(x)}{\partial x} = \begin{cases} w_1 & \text{if } w_1^T x + b_1 > w_2^T x + b_2 \\ w_2 & \text{if } w_1^T x + b_1 < w_2^T x + b_2 \end{cases}$$

Drawbacks:

- Each MaxOut layer requires **twice** the number of parameters as a ReLU layer, and is therefore more computationally expensive - and usually never used.

II.2.j Softmax

The Softmax function is defined as:

$$\mathbb{R}^K \to [0,1]^K$$

$$\text{Softmax}(x)_i = \frac{e^{x_i}}{\sum_{j=1}^K e^{x_j}}$$

Main features:

- Usually not used as a regular activation function, but in very specific cases: To make the logits of the network to be interpretable as probabilities, for example in the case of multi-class classification. Another example is to sharpen weights of the attention mechanism in transformers.
- It projects the input to the range $[0, 1]$, and can be interpreted as the probability of the input belonging to the i-th class.
- It magnifies the differences between the values of the input, and can be used to make the network more confident in its predictions - big numbers get bigger, and small numbers get smaller.
- It is differentiable, and has a simple derivative:

  $$\frac{\partial \text{Softmax}(x)_i}{\partial x_j} = \text{Softmax}(x)_i(\delta_{ij} - \text{Softmax}(x)_j)$$

  where δ_{ij} is the Kronecker delta -

  $$\delta_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}$$

- In order to avoid numerical instability, the input to the Softmax function is usually shifted by a constant value:

  $$\text{Softmax}(x)_i = \frac{e^{x_i - \max(x)}}{\sum_{j=1}^K e^{x_j - \max(x)}}$$

II.3 Loss functions

Loss functions represent the goal of training a neural network. They are the guidelines for the weights update, and should be tailored carefully to the task at hand. While some loss functions might look simple at first glance, they could lead to very specific behaviors of the network and the features selected - that are much more complex than one could imagine. Loss values should almost always be either positive or negative, so the network could learn to minimize or maximize them, respectively. Also, a learning process could involve multiple loss functions, that are combined together, and the network is trained to minimize the sum of them - where some are more important than others. These we call "regularization terms", as they make the training process more difficult, for the sake of achieving our desired behavior. It is crucial to understand that the sole goal of a training process of a neural network is to **minimize the loss value on the training set**. The performance on unseen data, such as the validation and test sets, is a byproduct of this process, and can only be achieved if we prevent the network from performing too well on the training set - a phenomenon called "overfitting". In this case, the network learns to capture features too specific to the training set (e.g. memorizing it) and not some general features, that could fit also data that it has never seen before. More on that at the optimization chapter (chapter IV).

Notes:

- The loss function **should always** result in a scalar: $L : \mathbb{R}^N \times \mathbb{R}^N \to \mathbb{R}$. That specific attribute is what allows us to learn the gradients of the single weight values, all the millions of them, in a way that doesn't require computing Jacobians. The variables in neural networks are always scalars, but just assembled in matrices and tensors. More on that at the TUM-article, that you could find on piazza, or chapter VII at the end of these notes.
- The loss function is differentiable (at least locally, like in the case of MAE $[L_1]$).
- We ALWAYS divide by N, the number of **residuals** in the loss function, to make the loss value independent of the size of the of numbers that are being compared. This is crucial, as the loss value should be comparable between different models and different datasets. Pay attention that the number of residuals is not necessarily the number of samples in the dataset, e.g. in the case of semantic segmentation, it is the number of pixels from all the images in the batch all together. Also, if we don't divide the sum, the gradient could become very large, as a dependency of how big the input is, and that could lead to exploding gradients.
- You've seen in class the term "cost function". The deep learning literature is saturated with ambiguities and no default definitions to important terms. Therefore, we drop this term in this course completely and discuss ONLY the loss functions.

We will distinguish between two main types of loss functions: classification loss functions and regression loss functions.

II.3.a Classification loss functions

Binary Cross-Entropy - BCE

The binary cross-entropy loss function is defined as:

$$L_{BCE}(y, \hat{y}) = -\frac{1}{N}\sum_{i=1}^{N}(y_i \log_e(\hat{y}_i) + (1 - y_i)\log_e(1 - \hat{y}_i))$$

where y_i is the true label, and $\hat{y}_i$ is the probability to belong to label 1.

Note: the value of $\hat{y}_i$ should never be exactly 0 or 1, as the logarithm of 0 is undefined. To prevent this, we can clip the values of $\hat{y}_i$ to be in the range $[0 + \epsilon, 1 - \epsilon]$.

Categorical Cross-Entropy - CE

The categorical cross-entropy loss function is defined as:

$$L_{CE}(y, \hat{y}) = -\frac{1}{R * \cdot C} \sum_{i=1}^{R} \sum_{j=1}^{C} y_{ij} \log_e(\hat{y}_{ij}) = -\frac{1}{R} \sum_{i=1}^{R} \log_e(\hat{y}_i)$$

where y_{ij} is an indicator of 0 or 1 if the input i belongs to class j, and $\hat{y}_{ij}$ is the predicted probability it actually does. R is the number of residuals (e.g. number of instances in the batch, all the pixels in the batch, etc).

Here we employ what we call the "one-hot" encoding, where the true label is a vector of length C with a single 1 at the index of the true class, and zeros elsewhere. The predicted probabilities are in a vector of length C:

$$y = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \hat{y} = \begin{bmatrix} 0.1 & 0.8 & 0.1 \\ 0.9 & 0.1 & 0.0 \\ 0.0 & 0.0 & 1.0 \end{bmatrix}$$

Therefore, in practice **the number of residuals** doesn't contain the size of the one-hot encoding vector (number of classes), as we only consider the residuals of the true class - so we don't also divide by C and the loss function value is independent of the number of classes.

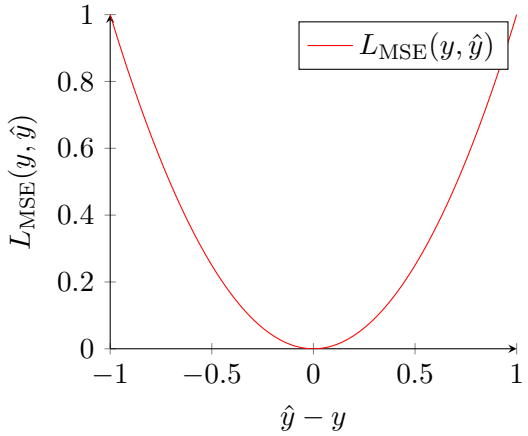
II.3.b Regression loss functions

Mean Squared Error - MSE

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

Also referred to as the "L2" loss function.

Calculates the "Euclidean" distance between the true and predicted values. The loss value is the average of the squared residuals. This loss function is sensitive to outliers, as it gives bigger weights to larger residuals (predictions that are farther from the true values), e.g.: a loss value of 0.1 is weighed with $0.1 \rightarrow 0.1^2 = 0.01$, while a loss value of 0.01 is weighed with $0.01 \rightarrow 0.01^2 = 0.0001$. The loss curve is defined as a parabola:

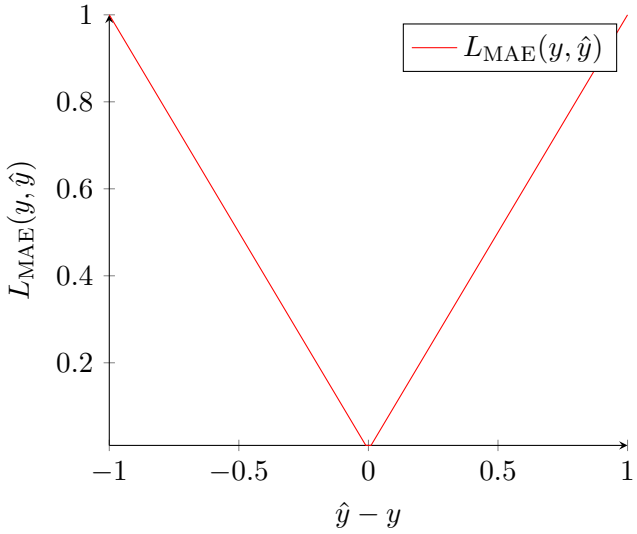


Mean Absolute Error - MAE

$$L_{MAE}(y, \hat{y}) = \frac{1}{N} \sum_{i=1}^{N} |y_i - \hat{y}_i|$$

Also referred to as the "L1" loss function.

Calculates the "Manhattan" distance between the true and predicted values. The loss value is the average of the absolute residuals. This loss function is less sensitive to outliers, as it gives equal weights to all residuals. The loss curve is defined as a "V" shape:



Note that it is not differentiable at $x = 0$, which could cause for some numerical instabilities, and adds a very small computational overhead, when calculating the gradient on a computer.

II.4 Linear Regression

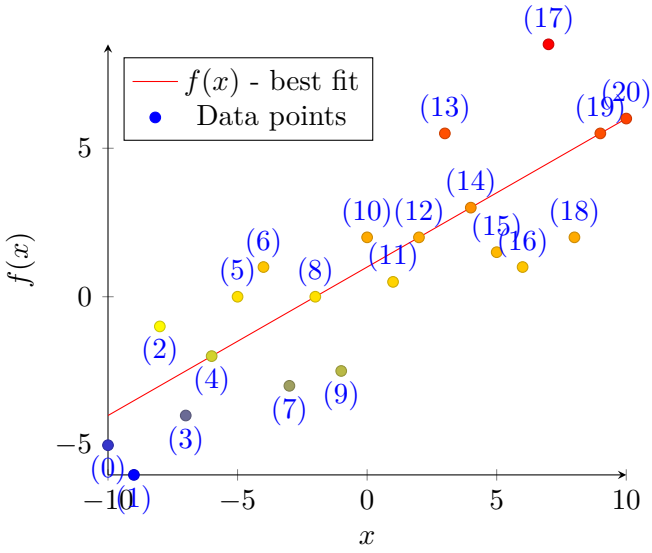


Figure II.3: Scatter plot with best fit line

Linear regression is a **linear** technique for modeling the relationship between different data points or variables. The simplest form of the regression equation with one dependent and one independent variables is defined by the formula:

$$Y = XW + b$$

where X represents the data, W is the weight, and b is the bias. The goal of linear regression is to find the best-fitting straight line through the data points, possibly in multiple dimensions. The most common method is to find the best-fitting line that minimizes the sum of the squared differences between the observed values and the predicted values. This is known as the **least squares criterion**:

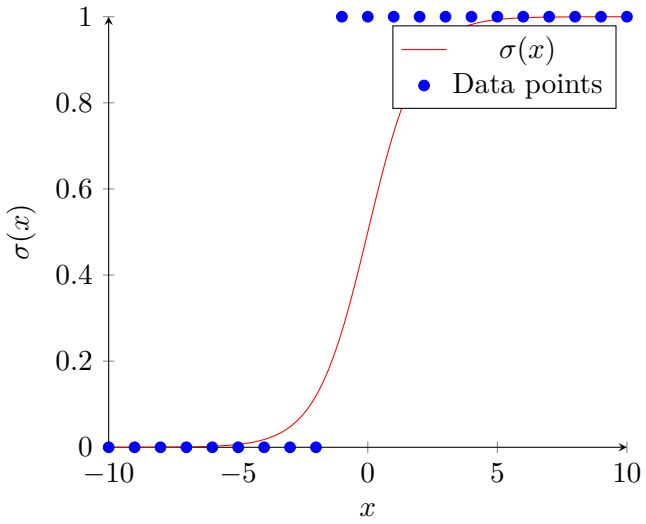
$$L(y_i|x_i, w) = \frac{1}{N}\sum_{i=1}^{N}(y_i - (w \cdot x_i + b))^2$$

The goal is to find the values of w and b that minimize this error. This can be done using the **closed-form** equation:

$$W = (X^T X)^{-1} X^T Y$$

This equation can be solved directly for W and b. That means, that if we can load all the data into our RAM, then we can solve the linear regression problem in one step, and with no need for iterative optimization. However, this is not always possible, and in practice, we often use iterative optimization methods such as **gradient descent** (subsection IV.1.a) to find the optimal values of W and b.

II.5 Logistic Regression



By using a very-very basic neural network, of simply one linear layer $f(x)$ followed by a non-linear activation function, we can now model a very common non-linear problem: the binary classification. In this setting, we do not try to fit a line to the data, but rather a curve that separates the two clusters or "classes". The linear layer is followed by the **sigmoid** activation function:

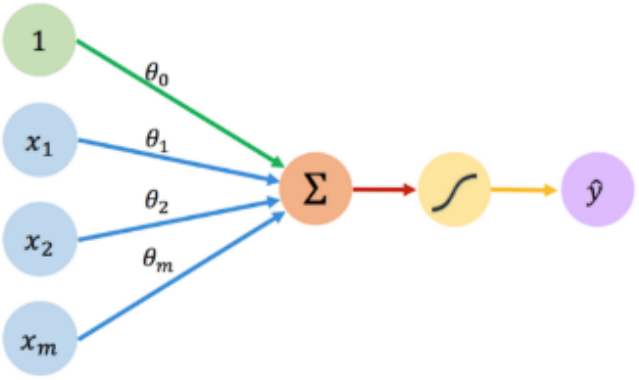
$$\hat{y} = \sigma(x) = \frac{1}{1 + e^{-f(x)}}$$

This function squashes the **logits** (the neurons in the output layer) into the range $[0, 1]$, and can be interpreted as the probability of the input belonging to the positive class. The loss function for binary classification is the **binary cross-entropy** ("BCE") loss:

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

Then, a threshold t, to establish a discrete boundary between the two classes:

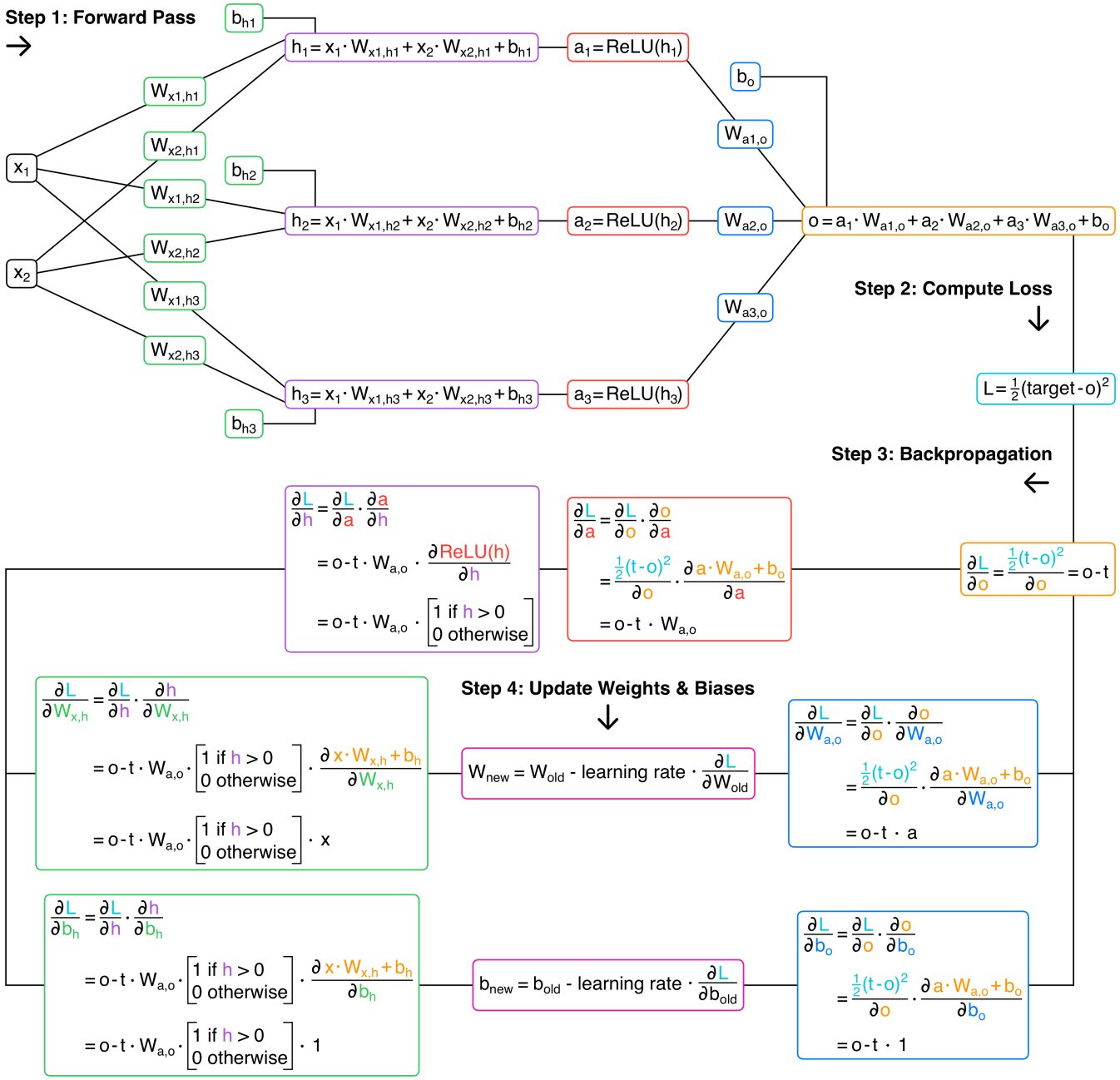
$$c_i = \begin{cases} 1 & \text{if } \sigma(f(x_i)) \geq t \\ 0 & \text{if } \sigma(f(x_i)) < t \end{cases}$$



II.6 Complete Schematic of a Fully Connected Neural Network

Now that we have covered Fully Connected Layers, Activation Functions, Loss Functions, and Backpropagation, we can put them together and build a complete fully connected architecture. For example:

- Two input neurons $[x_i]$, one hidden layer $[h_i]$ with ReLU $[a_i = ReLU(h_i)]$, and one output neuron
- MSE (Mean Squared Error) loss & GD (Gradient Descent) optimizer to update the weights and biases



Try to follow the four steps with some actual numbers to check if you understand how one training step works. Here are some sample values and the updated weights and biases:

$x_1 = 1, x_2 = 2, W_{x1,h1} = 0.1, W_{x2,h1} = 0.2, W_{x1,h2} = 0.3, W_{x2,h2} = 0.4, W_{x1,h3} = 0.5, W_{x2,h3} = 0.6, b_{h1} = 0.1, b_{h2} = 0.2, b_{h3} = 0.3, W_{a1,o} = 0.7, W_{a2,o} = 0.8, W_{a3,o} = 0.9, b_o = 0.1, target = 3.2, lr = 0.1$

New weights & biases: $W_{x1,h1} = 0.09, W_{x2,h1} = 0.18, W_{x1,h2} = 0.29, W_{x2,h2} = 0.37, W_{x1,h3} = 0.49, W_{x2,h3} = 0.57, b_{h1} = 0.09, b_{h2} = 0.19, b_{h3} = 0.29, W_{a1,o} = 0.69, W_{a2,o} = 0.78, W_{a3,o} = 0.87, b_o = 0.08$

II.7 Questions

1. Linear Regression

   a) Assume a linear problem of fitting with 5 billion points, each with 100 features. Is it feasible to find the optimal without a neural network? State if yes or no, and explain why.

   b) How can we transform a linear problem (with only affine layers) into a classification problem?

2. Fully Connected Layers

   a) Given the following layer $y = f(X, W, Z, R, T, A) = XW + ZR + T + A$, where $X_{N\times D}$, $W_{D\times M}$, $Z_{N\times D}$, $R_{D\times M}$, $T_{N\times M}$, $A_{N\times M}$ And a loss function $L : \mathbb{R}^N \times \mathbb{R}^M \to \mathbb{R} = \sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij}$. What is the gradient $\frac{\partial L}{\partial Z}$? What is the gradient $\frac{\partial L}{\partial A}$?

   b) Given batch of images of shape $(8\times3\times8\times8)$. How can we process them through a fully connected layer? What would be the shape of the weight matrix W in a case of a logistic regression and BCE as a loss function?

   c) Given batch of images of shape $(8 \times 3 \times 8 \times 8)$, a fully connected network (FCN) and a task to detect eyes of people in the images. Name two disadvantages for trying to solve the task with the current setup.

   d) Given two affine layers:
   $$y_1 = f_1(X, W_1, B_1) = XW_1 + B_1$$
   and
   $$y_2 = f_2(y_1, W_2, B_2) = y_1W_2 + B_2$$
   Show that it could be described as a single affine layer.

3. Activation Functions

   a) What is the purpose of the activation functions in a neural network?

   b) Give two advantages of the Tanh function over the Sigmoid function.

   c) Explain the vanishing gradient problem, and describe one method that could help us solve it.

   d) We've learned that the sigmoid function can cause the "vanishing gradient" problem. Therefore, explain why is the sigmoid function is still sometimes used on the logits (the output of the last layer).

   e) Assume the LeakyReLU activation function. If we take an $\alpha < 0$, what property of the function would we lose?

   f) The Softmax function could suffer from numerical instability, given the logits. Show that reducing the maximum value of the logits from each one of the values in the vector would not change the output of the function:
   $$\text{Softmax}(x - \max(x))_i = Softmax(x)_i$$

4. Loss Functions

   a) Which property of loss functions allows us to perform backpropagation without computing Jacobians?

   b) Assume that you're using the MSE loss function to compare to batches of images, of shape $(4 \times 3 \times 8 \times 8)$, what would be the value of N, that we should divide the loss value by?

c) In depth estimation, where we predict the depth of each pixel in an image, it was found that the loss for the majority of pixels is very small, while for some few pixels it is very large. What would be a better loss function to use in this case out of [MAE, MSE, BCE, CE], and why?

d) What could cause numerical instabilities in the BCE and CE functions? How could we solve that?

e) Why is the MAE loss function could be still used, although it is not differentiable at $x = 0$?

f) Assume that in the first iteration of training, some the logits are values above 1000, while the ground-truth values are in the range of [0, 1]. If we're using the MSE loss function - what optimization problem should we expect to observe?

g) Let's use the CE loss function for a task of classification of 100 classes. What is the expected loss value after the first iteration, and why?

h) BCE: how many neurons are at the output layer?

i) BCE: why do we multiply the result by -1?

j) Why don't we multiply by -1 in MSE or MAE?

k) You are given a neural network for a classification task with 4 classes and the CE as a loss function. The batch size is 1000. After the very first iteration of training, what is the expected loss value?

II.8 Answers

1. Linear Regression

   a) No, it is not feasible. Linear regression has a closed-form solution, but it requires loading to the RAM a matrix of size $X_{10^9 \times 100}$ and also inverse it, which is impossible. Therefore, we would have to use an iterative training scheme using batches through a neural network.

   b) Add a sigmoid function at the end of the network, and use one of the cross-entropy loss functions.

2. Fully Connected Layers

   a) $\frac{\partial L}{\partial Z} = \frac{\partial L}{\partial y} \cdot R^\top$, $\frac{\partial L}{\partial A} = 1_{N \times M} \odot \frac{\partial L}{\partial y}$, where $\cdot$ is matrix multiplication, and $\odot$ is element-wise multiplication.

   b) We could flatten the images, each to a vector of size $3 \times 8 \times 8 = 192 \rightarrow X_{8 \times 192}$, and the weight matrix would be of size $W_{192 \times 1}$.

   c) i. FC layers extract only global features, while the task requires local features.

      ii. The number of pixels (resolution) is too small, which means a lack of fine details.

   d) $y_2 = f_2(f_1(X, W_1, B_1), W_2, B_2) = f_2(XW_1 + B_1, W_2, B_2) = (XW_1 + B_1)W_2 + B_2 = X(W_1W_2) + B_1W_2 + B_2$

3. Activation Functions

   a) Activation functions introduce non-linearity to the network, and allow the network to learn complex patterns in the data.

   b) 1. The Tanh function is zero-centered, while the Sigmoid function is not. 2. The Tanh function has a larger maximum gradient than the Sigmoid function.

c) The vanishing gradient problem is a phenomenon where the gradients become exponentially smaller as they propagate "up the stream", leading to a scenario where the weights of the shallow layers are updated much more slowly or not at all. One method to solve it is to use skip connections, that allow "a highway for the gradients" to flow through the network.

d) The sigmoid function is still used on the logits, as it projects the logits to the range $[0, 1]$, and can be interpreted as the probability of belonging to a class in binary classification.

e) If we take an $\alpha < 0$, we would lose the zero-centered property of the function, as all the values would be non-negative.

f) $\text{Softmax}(x - \max(x))_i = \frac{e^{x_i - \max(x)}}{\sum_{j=1}^{K} e^{x_j - \max(x)}} = \frac{e^{x_i} e^{-\max(x)}}{\sum_{j=1}^{K} e^{x_j} e^{-\max(x)}} = \frac{e^{-\max(x)} e^{x_i}}{e^{-\max(x)} \sum_{j=1}^{K} e^{x_j}} = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}} = \text{Softmax}(x)_i$

4. Loss Functions

a) The property that allows us to perform backpropagation without computing Jacobians is that the loss function is a scalar function $L : \mathbb{R}^N \times \mathbb{R}^M \to \mathbb{R}$ and that the variables in neural networks are always scalars, but just assembled in matrices and tensors. Eventually, we aim to find $\frac{\partial L}{\partial v}$, where v is a single scalar variable, $v \in \mathbb{R}$.

b) $N = 4 \times 3 \times 8 \times 8 = 768$, as we divide by the number of residuals in the loss function.

c) The MAE loss function would be better, as it is less sensitive to outliers, and would fit the majority of the residuals better.

d) Numerical instabilities could be caused by the logarithm of 0 in the BCE and CE functions. We could solve that by clipping the values of the logits to be in the range $[0 + \epsilon, 1 - \epsilon]$.

e) The MAE loss function could still be used, although it is not differentiable at $x = 0$, as we assume local differentiability in the loss function, which is the case for almost all cases.

f) The gradients would be HUGE, so we should expect to observe the "exploding gradient" problem, where the update step would be too large, and the network would diverge.

g) The expected loss value after the first iteration would be $-\log(0.01) = 4.6$, as the network would be initialized with random weights, and the expected value of the loss function is $-\log(1/C)$, where C is the number of classes.

h) The number of neurons at the output layer would be 1, as we have a binary classification problem.

i) We multiply the result by -1 to make the loss value positive, as $\forall 0 < x <= 1 : -\log(x) \geq 0$.

j) We don't multiply by -1 in MSE or MAE, as the loss value should be positive, and the network should learn to minimize it.

k) $-\ln(0.25)$

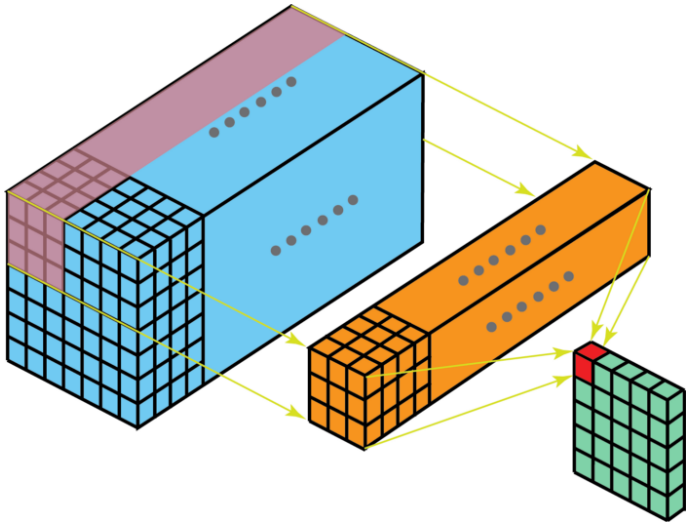
III Convolutions

Introduction Video: Link

While Fully Connected layers were the main linear building blocks in the early days of deep learning in computer-vision, they have been replaced by Convolutional layers, that introduce a few major advantages:

- While FC layers extract only **global** features, computer vision tasks require **local** features, of local windows, that are much more meaningful. More global look over the input image is achieved by deeper layer and their **receptive fields** (section III.3).
- While FC layers are fixed, and cannot take a different input size, convolutions can, as they only look at each local neighborhood at a time.
- Convolutions still represent a linear operation.
- Convolutions are also very efficient in the number of parameters and calculations.
- Convolutions are **translation Equivariant**:
  - Invariant: If the input changes, the output stays the same.
  - Equivariant: If the input changes, the output changes in the same way.

  meaning that if the input is translated, the output is also translated. Convolutions can easily find the same object in different parts of the input image, as they process local windows. However, note that they are not **rotation invariant** in general, but might be if the change is negligible.



III.1 Definition

The default convolution function is applied to an input with 3 key hyperparameters:

1. Kernel size k - the size of the kernel or "window" that is looked at a time. It has a height h and a width w, where usually $h = w = k$, but they could potentially be different. Usually chosen to be an **odd** number, so the window would have a center pixel.
2. Stride s - The amount of pixels the center of the window moves at the next step to the right, or down, once the row is over.
3. Padding p - the amount of pixels that are added to the edges of the image. Usually, the added value is zero, but could be any number. The padding allows us to preserve the spatial size of the image.

The convolutional layer is made up of filters. Each filter has a shape of (c_{in}, k_h, k_w), where c_{in} is the number of channels in the input image. The number of filters is c_{out}, the number of channels in the output tensor. That means, that each kernel takes at each time a spatial window, through all the channels, multiplies each entry with a corresponding weight, and then sums it all together to a single value:

$$\sum_{l=1}^{c_{\text{in}}} \sum_{i=1}^{k_h} \sum_{j=1}^{k_w} x_{l,i,j} w_{l,i,j}$$

The output is a tensor of shape $(c_{out}, h_{out}, w_{out})$, where h_{out} and w_{out} are the height and width of the output tensor. Those spatial values are calculated as follows:

$$H_{\text{out}} = \left\lfloor \frac{H_{\text{in}} + 2p - k_h}{s} \right\rfloor + 1$$

$$W_{\text{out}} = \left\lfloor \frac{W_{\text{in}} + 2p - k_w}{s} \right\rfloor + 1$$

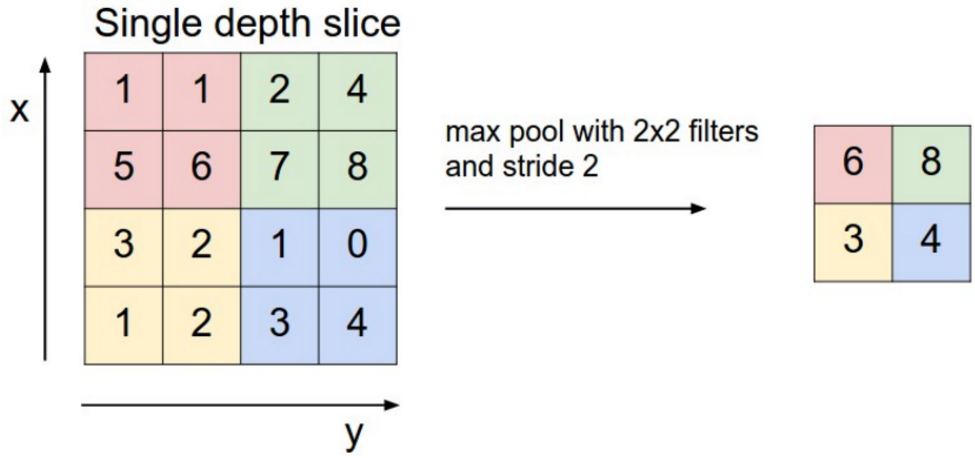
Note, that not all combinations are valid - meaning, that if not careful with the choice of these parameters, not all windows would be calculated. A window that doesn't fit into the input, will be simply dropped.

Popular trios:

- $k = 1, s = 1, p = 0$ - Pointwise convolutions - process only a single pixel across its channels. Keeps the spatial dimensions intact.
- $k = 3, s = 1, p = 1$ - The most common one. Has a small neighborhood, and it keeps the spatial size of the input ($h_{in} = h_{out}, w_{in} = w_{out}$
- $k = 3, s = 2, p = 1$ - The stride is doubled, so the output spatial size is half the input's.
- $k = 7, s = 4, p = 3$ - The stride is quadrupled, so the output spatial size is 1/4 of the input's.

III.2 Unique Convolutional Layers

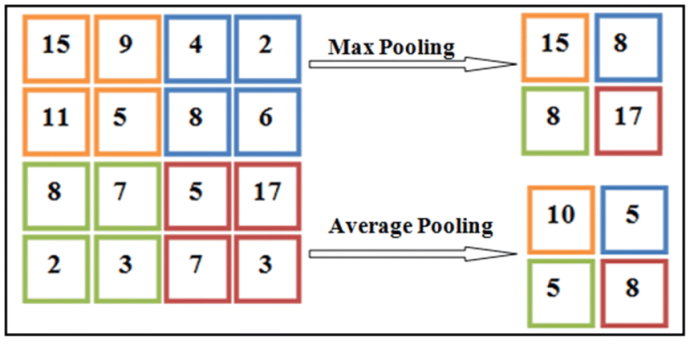
III.2.a MaxPooling



Max-pool is a layer that operates channel-wise (the kernel has a depth of 1), in contrast to the default convolution layer, where the kernel has the same depth as the input. Given a window, it returns the maximum value within it. Usually, it is used in order to reduce the spatial size of the input, while introducing non-linearity but no additional parameters. However, this layer doesn't necessarily reduce the spatial size, and could be used while maintaining the same spatial dimensions, e.g. in the case of the "inception layer" (section V.6). The most used case is as in the figure above, with a kernel size of 2×2, a stride of 2 and no padding.

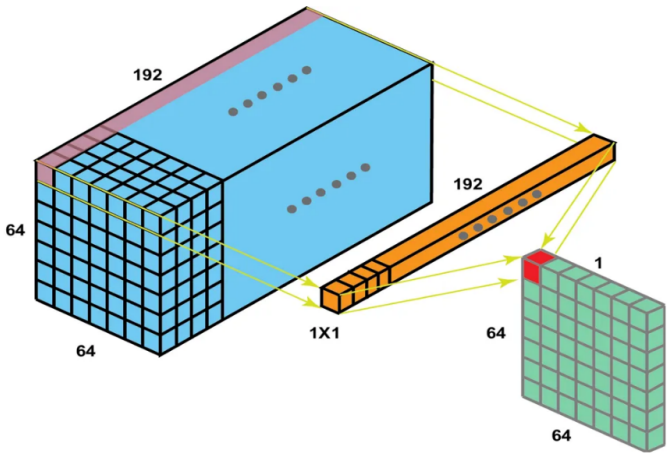
Note about the gradient: Only the entry (or multiple entries) in the window that hold the maximum value in that window, will get a derivative value of 1. The rest are killed with 0s.

III.2.b Average pooling



Similar to maxpool, but instead of returning the maximum value, it returns the average value of all entries of the window. Note, that this operation is linear, and does not introduce non-linearity. It is also normally used to reduce the spatial size of the input, with no additional parameters.

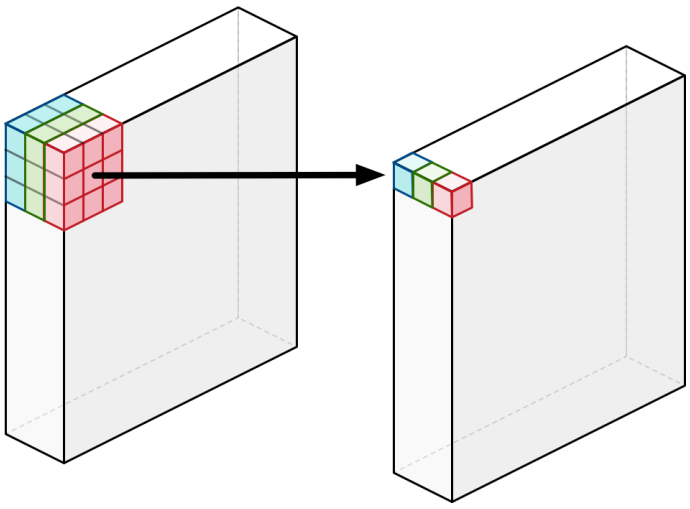
III.2.c 1×1 convolutions



Also called "Point wise convolution". This is a special case of the convolution layer, where the kernel is 1×1. This layer is used usually to reduce the number of channels in the input, while keeping the spatial size. This layer is also first introduced in the "inception layer" (section V.6), with the purpose of reducing the channels before the more expensive operations, such as $3 \times 3, 5 \times 5$ convolutions. With that, they managed to reduce dramatically the number of parameters and calculations, while maintaining the performance to some extent.

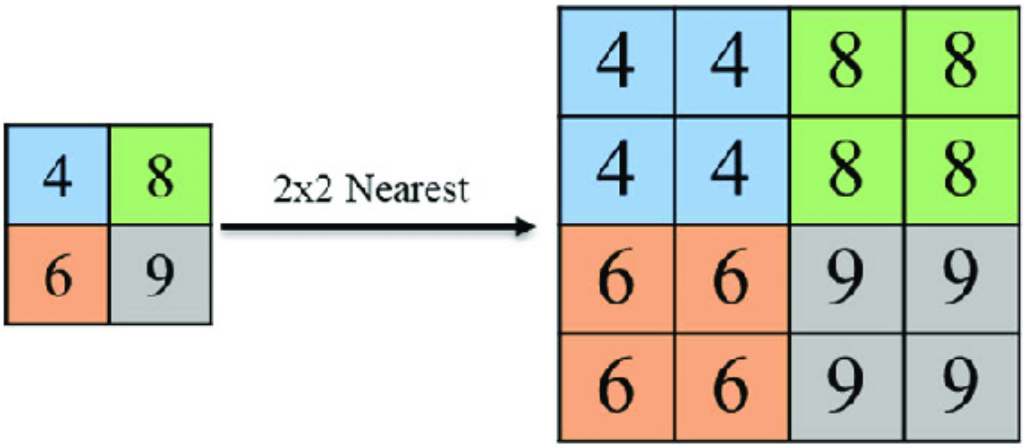
Note, that this layer doesn't extract spatial features, as it has no neighborhood, and its receptive field is the same as the layer before. Also, it is mentioned somewhere that this layer adds non-linearity. This is of course NOT TRUE, but one could use this layer, followed by some non-linear activation, since it is quite cheap.

III.2.d Depth-wise Convolutions



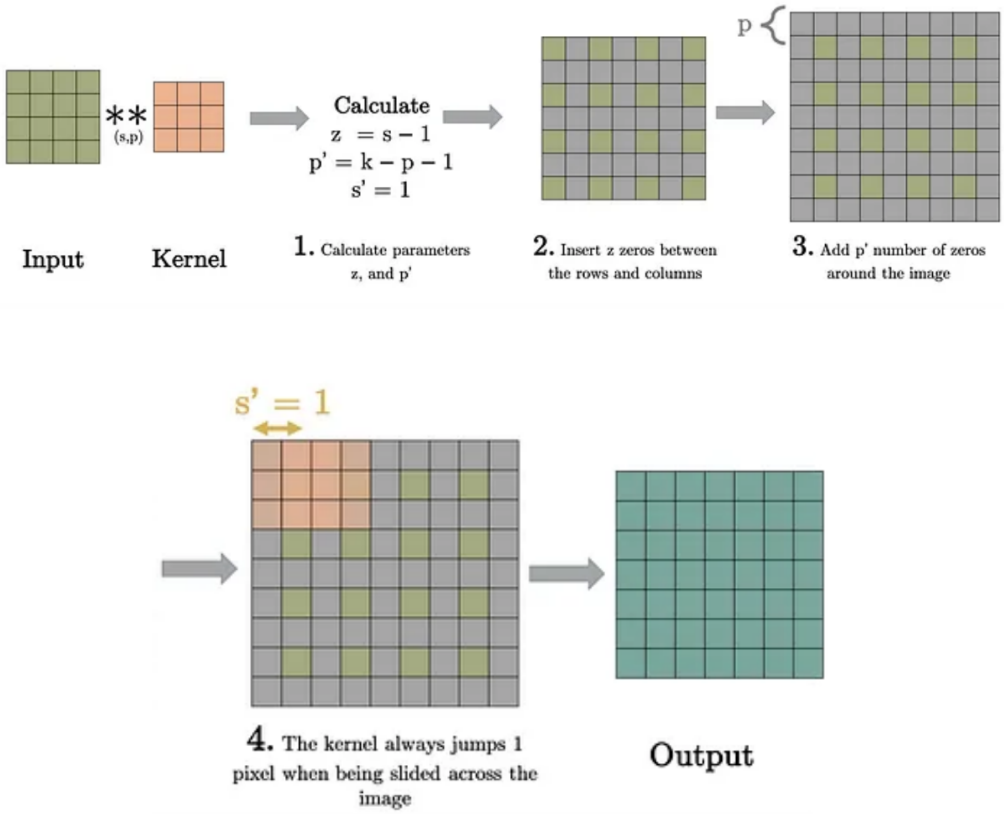
This is a special case of the convolution layer, where we take a single kernel of depth 1 for each channel. It is common in modern visual-transformers architectures.

III.2.e Upsample



A linear layer that introduces no additional parameters, to increase the spatial size of the input. Usually using a predefined algorithm, such as nearest neighbor, bilinear or bicubic.

III.2.f Transpose Convolutions



Unlike the upsampling layer, the transposed convolution introduces additional parameters in the form of kernels. While being more capable and able to adjust to the task at hand, it is also more expensive. Note that it is not the same as "deconvolution" layers.

III.3 Receptive Fields

The receptive field is the number of pixels in the network's input, which an intermediate pixel in some hidden layer is affected by. The spatial extent of the connectivity of a convolutional filter. What is the relationship between the input and the output? For example: 3x3 receptive field: 1 output pixel is connected to 9 input pixels. **Note**, that we do not consider the channels dimension - we only care about the spatial context.

To calculate the receptive field on a pixel in the intermediate layer l, we use the following formula:

$$r_l = r_{l-1} + (\prod_{i=1}^{l-1} s_i) \cdot (k_l - 1) \; for \; l \geq 2$$

Where s is the stride and k is the kernel size. Also, r_1 is the kernel size of the first layer. **Note** that we start from the input and go deeper, and not from the requested layer backwards.

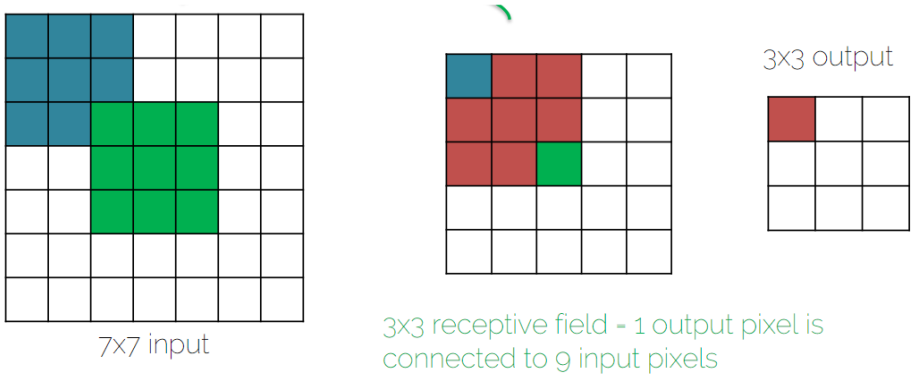
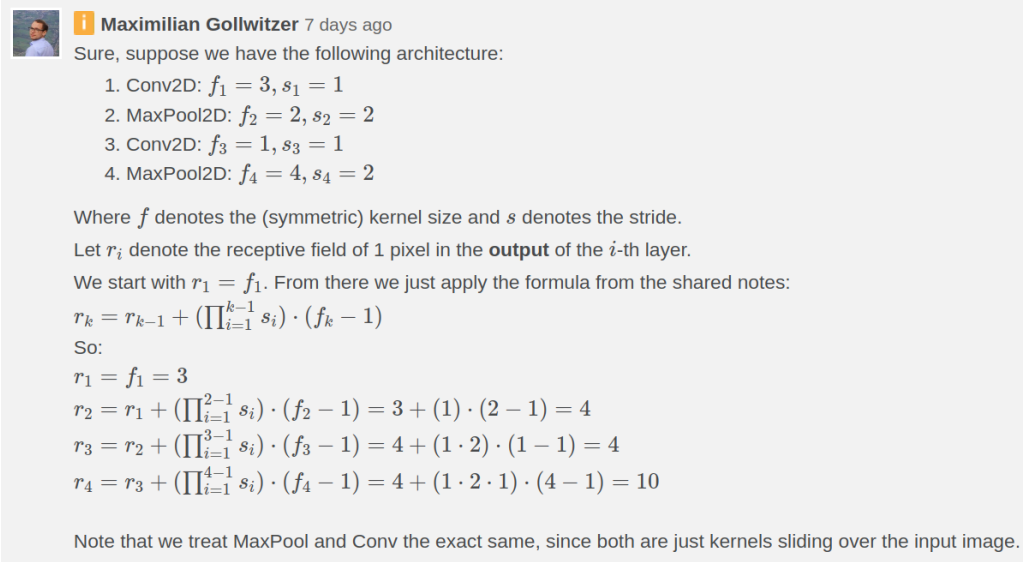


Figure III.1: Example: $r_2 = r_{2-1} + (\prod_{i=1}^{1} s_i) \cdot (k_2 - 1) = 3 + 1 * (3 - 1) = 5$
$r_3 = r_2 + (\prod_{i=1}^{1} s_i) \cdot (k_2 - 1) = 5 + (1 * 1) * (3 - 1) = 7$



Maximilian Gollwitzer 7 days ago

Sure, suppose we have the following architecture:

1. Conv2D: $f_1 = 3, s_1 = 1$
2. MaxPool2D: $f_2 = 2, s_2 = 2$
3. Conv2D: $f_3 = 1, s_3 = 1$
4. MaxPool2D: $f_4 = 4, s_4 = 2$

Where f denotes the (symmetric) kernel size and s denotes the stride.

Let r_i denote the receptive field of 1 pixel in the **output** of the i-th layer.

We start with $r_1 = f_1$. From there we just apply the formula from the shared notes:

$r_k = r_{k-1} + (\prod_{i=1}^{k-1} s_i) \cdot (f_k - 1)$

So:

$r_1 = f_1 = 3$

$r_2 = r_1 + (\prod_{i=1}^{2-1} s_i) \cdot (f_2 - 1) = 3 + (1) \cdot (2 - 1) = 4$

$r_3 = r_2 + (\prod_{i=1}^{3-1} s_i) \cdot (f_3 - 1) = 4 + (1 \cdot 2) \cdot (1 - 1) = 4$

$r_4 = r_3 + (\prod_{i=1}^{4-1} s_i) \cdot (f_4 - 1) = 4 + (1 \cdot 2 \cdot 1) \cdot (4 - 1) = 10$

Note that we treat MaxPool and Conv the exact same, since both are just kernels sliding over the input image.

Note: For fully connected layers, the receptive field is the ENTIRE image. Since each output neuron of that layer is connected to each neuron of the previous one, where each layer represents the entire output, whether it is down-scaled or upscaled.

III.4 Handcrafted Kernels

Important: For some reason, these keep popping up in the exams. You should know them **by heart**, or just decide to skip them in advance. The most common ones are:

- Edge detection:
  - $\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$
  - Sobel - vertical and horizontal filters: $\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$
- Blurring:
  - Box mean: $\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$
  - Gaussian blur: $\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$
- Sharpen:
  - $\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$

III.5 Questions

1. How can we represent a FC layer with 5 output neurons with a convolutional layer, over an image in a batch of size $4 \times 3 \times 8 \times 8$? And with 1x1 Convolution?
2. Given a 1×1 convolutional layer with input tensor of 10 channels, that outputs a tensor with 5 channels.
   a) Write the shape of the weight matrix.
   b) State the number of parameters in the layer.
3. Give two reasons to use a convolution or a pooling layer that reduce the spatial size of the input tensor.
4. Does reducing the spatial size throughout the network reduce the number of parameters in the down-the-stream convolutional layers?
5. Does reducing the spatial size throughout the network reduce the number of parameters in the down-the-stream FC layers?
6. State two differences between a convolutional layer and a pooling layer.
7. Assume a fully convolutional model for some task:
   a) Can we feed the model with images that are double the resolution of the original training set?
   b) Can we expect in such case the same performance?

8. Can we apply a pooling layer without reducing the spatial size of the input tensor?
9. Assume that when using maxpool with $(k = 2, s = 2, p = 0)$, where only a single entry in a given window holds the maximum value in that window. How many of the total pixels in the tensor would get a live gradient? What is the value of said gradient?
10. State one advantage and one disadvantage of a 1×1 convolution over a 3×3 convolution.
11. Why don't we use hand-crafted kernels (e.g. Sobel filter for edge detection) within our deep learning models?
12. In what technique, however, can we use hand-crafted kernels such as Gaussian blur?
13. Assume that we want to use a transformer to process an intermediate feature map (an output tensor of a convolutional layer), to learn meaningful relations between the pixels. How can we change the tensor to do that?

III.6 Answers

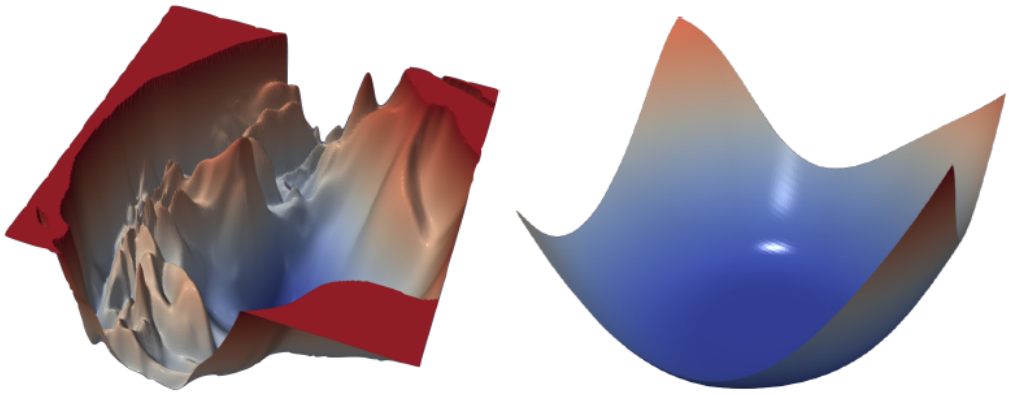
1. FC layers take ALL neurons in a single input from the batch. To do that with convolutions, we must have a kernel with $k_h = H_{in}, k_w = W_{in}$ and $k_c = C_{in}$. The weight matrix would be $5 \times 3 \times 8 \times 8$ (Number of kernels $\times$ $k_c \times k_h \times k_w$). The output tensor would be of shape $4 \times 5 \times 1 \times 1$. With a 1x1 Convolution, we can simply reshape the input tensor to $4 \times (3 * 8 * 8) \times 1 \times 1$.
2. a)
   - $w = 5 \times 10 \times 1 \times 1$
   - $b = 1 \times 5 \times 1 \times 1$

   b) 55.
3. Two reasons to reduce the spatial size of the input tensor are:
   - Compression enforces feature selection and extracting meaningful features.
   - Using smaller tensors throughout the network reduces the number of calculations, allowing us to use more layers for a greater capacity.
4. No - as long as the spatial size remains bigger than the convolutional kernel, the kernel sizes don't change, so the number of parameters stays the same.
5. Yes - reducing the spatial size throughout the network would result with smaller tensors, and therefore smaller FC layers, that take as input the entire previous layer.
6. Two differences between a convolutional layer and a pooling layer are:
   - A convolutional layer has learnable parameters, while a pooling layer doesn't.
   - Pooling layers work channel-wise (each channel independently), while default convolutional layers have the same depth as the input tensor.
7. a) Yes, fully convolutional networks can take varying sizes of inputs without any architectural changes, as long as it makes sense.

   b) No, convolutional layers are not invariant to scale changes.
8. Yes, we can apply a pooling layer without reducing the spatial size of the input tensor by using a relevant trio, e.g. (k=3, s=1, p=1), just like with convolutions.
9. Each window is of size 2×2, therefore has 4 entries. Only a single entry gets a gradient, and therefore $\frac{1}{4}$ of the total entries would have a live gradient, with a value of 1.

10. 1×1 vs 3×3:
    a) Advantage: Much cheaper in terms of parameters and calculations.
    b) Disadvantage: Smaller receptive field, so the scope of the layer is much narrower.
11. We don't use hand-crafted kernels within our deep learning models because they are not learnable, and thus not able to adapt to the task at hand.
12. We can use hand-crafted kernels such as Gaussian blur within our deep learning models by using them as the initial weights of the convolutional layers, and then updating them during training / use in data augmentation.
13. To use a transformer to process an intermediate feature map, we can change each instance in the batch to a 2D tensor, by transposing the channels to the last dim, and flattening the spatial size: $(B \times C \times H \times W) \rightarrow (B \times H \times W \times C) \rightarrow ((B \times (H * W) \times C)$. Now each pixel, across the channels, is a vector token for the transformer head.

IV Optimization

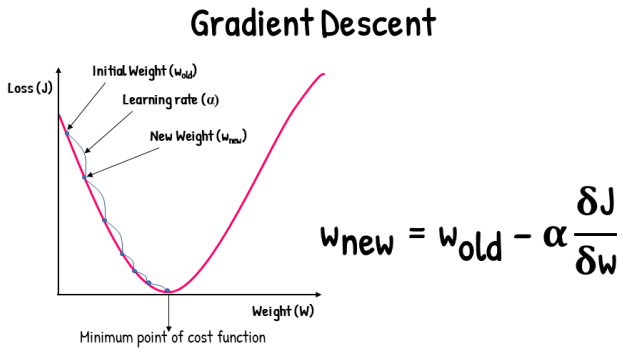


(a) ResNet-110, no skip connections (b) DenseNet, 121 layers

Since we deal with big-data and non-linear problems, we use iterative optimization algorithms to find the best solution. Neural networks consist of many different functions all together, and therefore their optimization problem is usually non-convex. This means that off-the-shelf basic neural networks would face a loss surface that has many different minimas. However, it is common to assume that any local minima is good enough as a solution, but to that end - we always aim to have the best solution and performance. In this section we will discuss the different approaches that exist for solving iterative optimization problems, and see why some of them are better than others, practically vs. theoretically.

IV.1 Optimization algorithms

IV.1.a Gradient Descent



Gradient descent (GD) is the basic optimization algorithm, on which all modern deep learning optimization problems are based. It follows a very basic idea: For each variable θ in the model, calculate the gradient over **the entire training-set**, and then perform an update on the variable θ in the direction of the negative gradient. This is done iteratively until the loss function converges to a minimum. The update rule is as

follows:

$$\theta_{t+1} = \theta_t - \alpha \frac{\partial L}{\partial v_t}$$

Where v_t is the variable at **epoch** t, α is the learning rate, and $\frac{\partial L}{\partial \theta_t}$ is the gradient of the loss function L with respect to the variable θ_t.

Why go in the direction of the negative gradient? Because the gradient points in the direction of the steepest ascent, and we want to minimize the loss function, so we go in the opposite direction.

Notes:

- Gradient descent is a first-order optimization algorithm, meaning that it only uses the first derivative of the loss function.
- In gradient descent, we perform the optimizer step only at the end of the epoch, which is only when we've iterated over the entire training set.
- This could be done in batches. Since datasets are usually too large to handle, we split them into smaller batches and use gradient accumulation - we accumulate the gradients in some buffer and at the end of the epoch, we divide by the number of iterations and only then perform the optimizer step.
- While gradient descent calculates the most accurate gradient, it has a few major drawbacks:
  - If we perform the optimizer step only at the end of the epoch, it each single step could take a long time to compute - the model will never converge in reasonable time.
  - Calculating the gradient over the entire training set prevents the approximation noise that comes from doing the optimizer step over a small batch, which is actually a good regularization power, that helps avoid saddle points. Therefore, GD tends to underfit.
  - Calculating the gradient over the entire training set is very computationally expensive, and therefore it is not practical for large datasets.

Python implementation of the GD algorithm:

```
def GD(model, loss_fn, optimizer, training_dataloader, epochs, lr):
    for epoch in range(epochs):
        loss = 0
        optimizer.zero_grad()
        for iter, batch in enumerate(training_dataloader):
            x, y = batch
            y_pred = model(x)
            loss = loss_fn(y_pred, y) / len(training_dataloader)
            loss.backward()
        optimizer.step()
```

IV.1.b Stochastic Gradient Descent

Stochastic gradient descent (SGD) is a variation of the gradient descent algorithm, that aims to solve the problems that come with the basic GD. While the original definition consists of using a batch size of 1, in practice we use a small batch size from the training set and perform the optimizer at **each iteration**. This way, we can perform the optimizer step many times in a single epoch, and therefore the model will converge faster, although it has a much noisier gradient.

The update rule is exactly the same as in GD, but we perform the optimizer step at each iteration:

$$\theta_{t+1} = \theta_t - \alpha \frac{\partial L}{\partial \theta_t}$$

at **iteration** $t+1$.

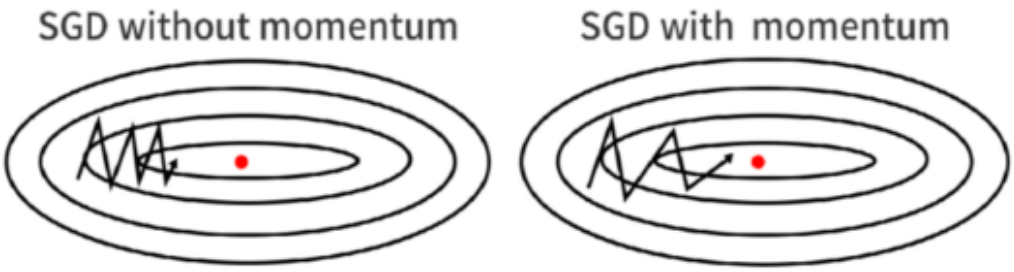
Notes:

- SGD is a first-order optimization algorithm, meaning that it only uses the first derivative of the loss function.
- The original definition of SGD is when the batch size is 1, but in practice, we use a small batch size (e.g 4, 8, 16, 32, 64, etc.).
- SGD is much faster than GD.
- "Noisier steps" comes from the fact that SGD only approximates the global gradient on a small batch, which cannot represent the real distribution of the data. Therefore, features that are learned on a current sample, might not be relevant for a sample from the next batch. However, this could be a good thing, as it helps avoiding saddle points. The bigger the batch size is, the lower the noise is.
- Implementation-wise, the ONLY difference between GD and SGD is that in SGD we perform the optimizer step at each iteration, and not at the end of the

Python implementation of the SGD algorithm:

```
def SGD(model, loss_fn, optimizer, training_dataloader, epochs, lr):
    for epoch in range(epochs):
        for iter, batch in enumerate(training_dataloader):
            optimizer.zero_grad()
            x, y = batch
            y_pred = model(x)
            loss = loss_fn(y_pred, y)
            loss.backward()
            optimizer.step()
```

IV.1.c SGD with Momentum



SGD with momentum is a variation of the SGD algorithm, that aims to solve the problems that come with the basic SGD. The idea is to add a momentum term to the update rule, that will help the optimizer to converge faster. The update rule is as follows:

$$m_{t+1} = \beta m_t - \alpha \frac{\partial L}{\partial \theta_t}$$

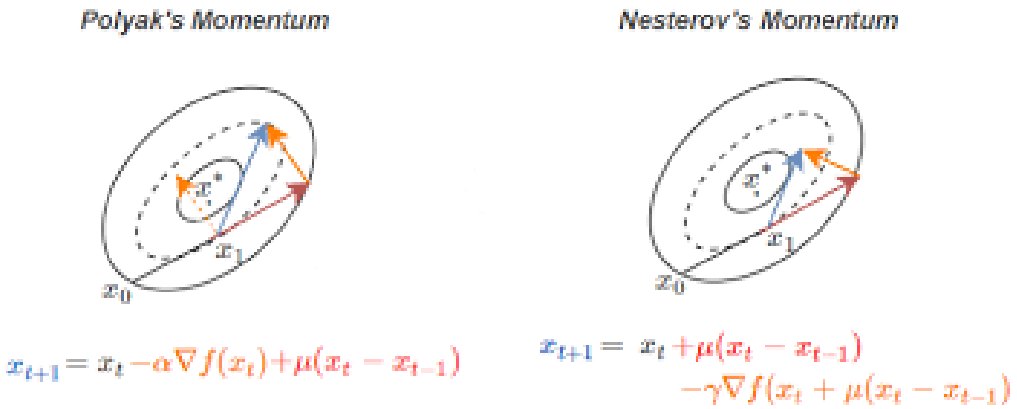
$$\theta_{t+1} = \theta_t + m_{t+1}$$

Where m_t is the momentum term at iteration t and β is the momentum coefficient (usually 0.9).

Notes:

- SGD with momentum is a first-order optimization algorithm.
- m_0 is usually initialized to 0.
- The momentum is being increased or decreased iteratively by calculating the momentum from one iteration to the next. It is similar, but not exact, to the "exponential moving average" concept.
- When the "slope" is steep, the velocity will increase very fast, allowing us to overshoot saddle-points.

IV.1.d Nestrov Momentum



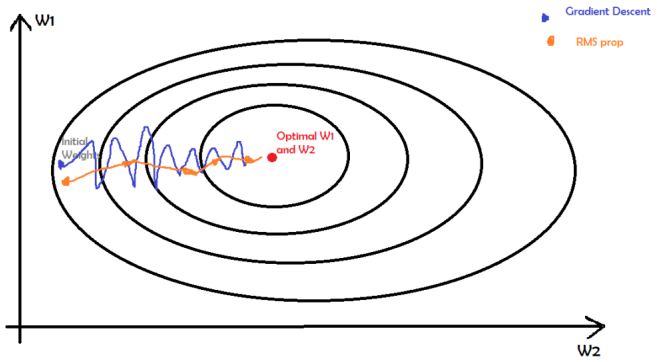
The Nesterov Momentum optimizer (NAG) improves on standard momentum by anticipating the direction of the update, computing the gradient at a point slightly ahead ("Look Ahead") of the current position. This could result in faster convergence and better training performance for deep learning models.

The update rule is as follows:

$$\tilde{\theta}_{t+1} = \theta_t + \beta m_t$$
$$m_{t+1} = \beta m_t - \alpha \nabla_v L(\tilde{\theta}_{t+1})$$
$$\theta_{t+1} = \theta_t + m_{t+1}$$

1. Compute the new point just ahead of the current point, given the current momentum.
2. Compute the new adjusted velocity m_{t+1}
3. Perform the actual update to the variable θ at time $t+1$

IV.1.e RMSprop



This algorithm follows the idea to divide the learning rate by the square root of the sum of the squares of the gradients, to mitigate big oscillations in the gradient. Since this is done for each variable in the gradient individually, it results in a different optimization step size for each one of them. The update rule is as follows:

$$v_{t+1} = \beta v_t + (1-\beta)[\nabla_\theta L \circ \nabla_\theta L]$$
$$\theta_{t+1} = \theta_t - \alpha \frac{\nabla_\theta L}{\sqrt{v_{t+1}} + \epsilon}$$

Where s_t is the velocity term at iteration t, β_2 is the momentum coefficient (usually 0.99), and ϵ is a small number to avoid numerical instability.

Notes:

- RMSprop is a **second-order** optimization algorithm, meaning that it uses the second moment of the loss function gradient.
- The idea is to divide the learning rate by the square root of the sum of the squares of the gradients, to mitigate big oscillations in the gradient.
- The optimization step is different for each variable in the gradient - hence it is called an adaptive learning rate.
- The squared gradients approximate the second moment of the gradient, which is the variance: Recall that the variance is
$$Var(X) = E[(X-\mu)^2]$$
We assume that the mean of all variables is $\mu = 0$. Therefore,
$$\nabla_\theta L\left(\theta_t\right)^2 = [\nabla_\theta L\left(\theta_t\right) \circ \nabla_\theta L\left(\theta_t\right)]$$

$$Var(\nabla_\theta L(\theta_t)) \sim E[\nabla_\theta L(\theta_t)^2] \sim \beta_2 \cdot v_t + (1-\beta_2)\left[\nabla_\theta L(\theta_t) \circ \nabla_\theta L(\theta_t)\right]$$

Which is accumulated iteratively. It accumulates a "running variance" of the gradients.

IV.1.f Adam - Adaptive Moment Estimation

Adam is a combination of RMSprop and SGD with momentum. It uses the squared gradients to scale the learning rate like RMSprop, and it uses the momentum term to avoid local minimas like SGD with momentum. The update rule is as follows:

$$\begin{aligned}
m_{t+1} &= \beta_1 m_t + (1-\beta_1)\nabla_v L \\
v_{t+1} &= \beta_2 v_t + (1-\beta_2)[\nabla_\theta L \circ \nabla_v L] \\
\hat{m}_{t+1} &= \frac{m_{t+1}}{1-\beta_1^{t+1}} \\
\hat{v}_{t+1} &= \frac{v_{t+1}}{1-\beta_2^{t+1}} \\
\theta_{t+1} &= \theta_t - \alpha \frac{\hat{m}_{t+1}}{\sqrt{\hat{v}_{t+1}+\epsilon}}
\end{aligned}$$

m represents the running mean of the first-moment, then mean of the gradients, and v stands for the running variance - for each single entry of the gradient matrix.

Notes:

- Adam is a **second-order** optimization algorithm
- The idea is to combine the best of both worlds - RMSprop and SGD with momentum.
- **Bias correction**: The Adam optimizer uses a bias correction term to correct the bias of the first and second moments. It was first introduced with Adam, and therefore doesn't exist in RMSprop, although it should. The idea is that at the first iteration, the variables m, s are initialized to 0, and therefore they are biased towards 0. To correct this bias, we divide them by the term $1-\beta^{t+1}$, where t is the iteration number, and here is used as the **exponent**. As the number of iterations increases, the correction terms vanish, but at the very first ones - we neglect the m, s terms, and therefore the optimizer step is much more meaningful:

$$m_1 = \beta_1 m_0 + (1-\beta_1)\nabla_\theta L$$

$$\hat{m}_1 = \frac{m_1}{1-\beta_1} = \frac{\beta_1 m_0 + (1-\beta_1)\nabla_\theta L}{1-\beta_1} = \nabla_\theta L$$

IV.1.g Newton's Method and its Variants

While Stochastic gradient descent and its variants show great capabilities, they are a leg behind in theory behind some others. Newton's method is a second-order optimization algorithm, that uses the second derivative of the loss function to find the optimal solution, in much fewer iterations than SGD. It follows Newton's-Raphson method, which is a root-finding algorithm that uses the second derivative of the function to find the root (e.g. $y = f(x) \rightarrow y = 0$) of the function. The update rule is as follows:

Newton-Raphson method:

$$x_{t+1} = x_t - \frac{f(x_t)}{f'(x_t)}$$

But here we want to find the direction of the gradient, so we're looking for the derivative of the gradient, which is the second-derivative:

For a single variable x:

$$x_{t+1} = x_t - \frac{f'(x_t)}{f''(x_t)}$$

For a matrix θ:

$$\theta_{t+1} = \theta_t - H^{-1}\nabla_\theta L(\theta_t)$$

Where H is the Hessian matrix (matrix of second derivatives) according to the loss function, and H^{-1} is its inverse.

Notes:

- Newton's method is a **second-order** optimization algorithm.
- This method is NOT FASTER than SGD, but will converge in fewer iterations, as the **inverse** Hessian matrix is really expensive to compute.
- So why isn't it used in practice? the Newton's method requires the entire training set as input to compute the inverse Hessian matrix, which is super expensive in memory and time.
- There are some variants of the Newton's method that are used in practice, that approximate the Hessian matrix and therefore are much faster. The most common one is the L-BFGS algorithm. However, they also still require the entire training set as input, and therefore are not practical for large datasets and deep learning.

IV.1.h Second Order confusion

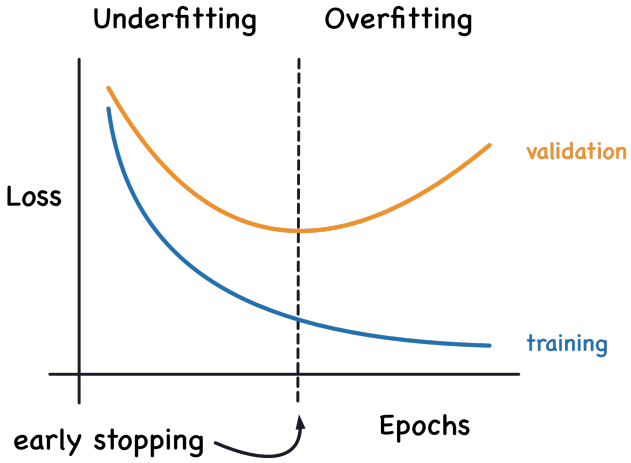
In literature, you'll find a lot of inconsistencies about this term. Some claims about Adam, RMSprop and the newton method being **second order methods**, and some say that only the newton method is.

- **It doesn't really matter. It's not important. Don't break your head over it. Simply, until stated otherwise, in this course - all the three above are second order.**
- Adam and RMSprop are a second-order gradient-descent-based optimizer (second moment).
- Newton's method is a second-order optimization method (second derivative).

Both cases are shorty referred to as "second order" methods.

IV.2 Optimization problems and solutions

IV.2.a Overfitting vs underfitting



As explained before, the sole goal of training neural networks - is to minimize the loss function and the generalized performance on unseen data is only a byproduct of that. However, that imposes a problem: if the model is too capable, it could fit the training data **too well**. There are plenty of reasons for such a behavior, such as:

- The model is too complex, and therefore has too many parameters. Instead of learning general patterns, it memorizes the training data.
- Too little data, and therefore the model has no way to learn the general patterns, but only the specific that fit the small dataset.
- Not stopping early enough, so after learning relevant, more general, features - the model learns the noise in the data.

That behavior is called in the literature "overfitting". How can we detect it? By looking at the validation loss. If the "generalization gap", the difference between the training loss and the validation loss, is too big - the model is overfitting. Usually, by monitoring those losses during training, we would see that while the training loss is decreasing, the validation loss is either increasing (diverging) or staying the same.

Solutions:

- Increasing the size of the training set: This is NOT a valid solution, as the creation of a dataset is a very difficult and expensive task, and we should try and work with the limited resources available. A method that could create more data out of the existing one is called "data augmentation".
- Regularization: The go-to methods. Make training harder, so the network will learn the general patterns. Examples: Weight-decay, Dropout, data augmentation, etc.
- Early stopping: In case one doesn't save checkpoints during training, the model could be stopped when the generalization gap starts to grow. However, this is NOT a regularization term, as it doesn't make the training harder, but rather stops it.
- Hyperparameter tuning: The model might be too complex, and therefore we should try and reduce its capacity by tuning the hyperparameters.

On the other hand, a model might not be able to reduce the training loss to a sufficient level, where it solves the problem of the training data. This counter phenomenon is called "underfitting". That could occur for some of the following reasons:

- The model's capacity is too low, meaning it is either too small or uses a bad architecture, and therefore cannot learn the general patterns in the data.
- The model is not trained for enough epochs and therefore has not converged to a minimum.
- The model is using a very basic optimization algorithm, and gets stuck in a "saddle point", where the gradient is 0.

How to detect it? First, we could look at training loss. If it is still decreasing (along with the validation loss) when the model stops training - then it means it could have been trained for more epochs. Second, we could assess the accuracy of the model on the training data: check the classification accuracy, visualize the reconstructed image, etc. Undesirable results would mean that the model fails to complete its task to overfit the training data.

Solutions:

- Increasing the model's capacity: by changing the architecture, adding more layers, etc.
- Training for more epochs: The model might not have converged to a minimum yet.
- Using a better optimization algorithm: The model might be stuck in a saddle point, and therefore we should use a better optimization algorithm.
- Learning rate decay: The model might be oscillating ("overshooting") around the minimum, and therefore we should decrease the learning rate.

IV.2.b Vanishing and exploding gradients

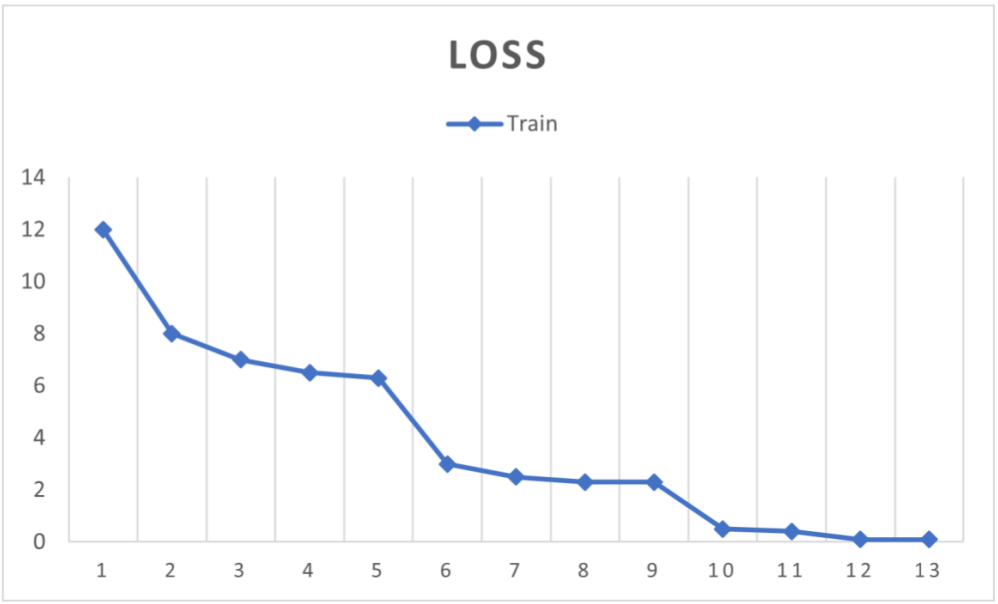
We've discussed in a previous section (subsection II.2.e) the problem of vanishing gradients and methods to solve it. On the other hand, exploding gradients is a problem that occurs when the gradients are too big, and therefore the optimizer step is too big, and the model diverges. This could happen for a few reasons:

- Recurrent cells: In RNNs, the gradients are being multiplied by the same matrix at each time step. If the matrix has an eigenvalue greater than 1, the gradients will explode.
- No normalization: If activations become very large after linear layers, then the gradients of the weights will be very large as well.
- Bad initialization: If the weights are initialized to very large values, the activations will also be very large, and that could affect the next layers' gradients.

Solutions:

- Gradient clipping: The idea is to clip the gradients to a certain value, so they won't explode. This is a very common method in RNNs, where the gradients are usually very large. Note: this method doesn't clip them for below, so it doesn't help with vanishing gradients.
- Normalization: The idea is to normalize the activations after each layer, so they won't become too large. Examples: Batch normalization, Layer normalization, etc.
- Weight initialization: Use known initialization methods, such as Xavier or Kaiming initializations, that will prevent the activations from becoming too large.

IV.2.c Learning rate scheduling



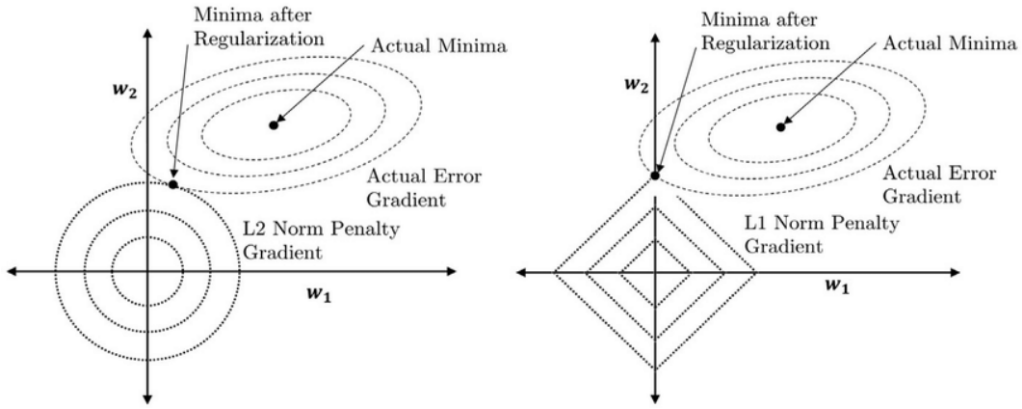
If you don't implement this crucial technique, stop doing deep learning and go do something else with your life. Learning rate scheduling is so critical, as the learning rate is the hyperparameter that affects the optimization process the most. The basic notion is to decrease the learning rate as the training progresses, so the optimizer step will be smaller and smaller, and the model will converge to a minimum. In addition, this method could also be used as regularization, if used correctly as described in the amazing paper of "super-convergence" (Link). Robbins and Monro have summarized (in 1951) the conditions for **theoretical** convergence for a **strictly convex** function $F(x, \theta)$:

- α is the initial learning rate.
- $a_n \geq 0, \forall n \geq 0$
- $\sum_{n=1}^{\infty} \alpha_n = \infty$
- $\sum_{n=1}^{\infty} (\alpha_n)^2 < \infty$
- $a_n \propto \frac{a}{n}$, for $n > 0$

IV.2.d Regularization

Regularization techniques are common practices in deep learning to prevent overfitting, but also introducing constraints or secondary objectives to the optimization problem. Regularization methods usually make the training process harder, so the model will seek to learn more general patterns in the data, in order to overcome the added difficulty and keep brining the training loss down. Usually, we will observe that the training loss will be higher, but the validation will be lower, hence the generalization gap would be minimized.

L1 and L2 regularization + Weight decay



A basic technique of adding a penalty term to the loss function, that tries to minimize the weights themselves, namely pushing them towards zero. If it succeeds, the model will not be able to calculate anything. The most common regularization terms are the L1 and L2 regularization, which are defined as follows:

$$L_1 : \lambda \sum_{i=1}^{\#\Theta} |\theta_i|$$

$$L_2 : \frac{1}{2}\lambda \sum_{i=1}^{\#\Theta} \theta^2$$

Where λ is the regularization coefficient, and w_i is the i-th weight in the model. When chosen, for example, in the L_2 regularization, they are added to the loss function, and the optimizer will try to minimize them as well:

$$L_{reg} = L + \frac{1}{2}\lambda \sum_{i=1}^{\#\Theta} \theta_i^2$$

$$\frac{\partial L_{reg}}{\theta_{l,i,j}} = \frac{\partial L}{\theta_{l,i,j}} + \lambda \theta_{l,i,j}$$

Where L is the original loss function. The L_1 regularization is also called "Lasso" regression, and the L_2 regularization is also called "Ridge" regression. Note the λ hyperparameter that controls the strength of the regularization term. If λ is too big, the model will not be able to learn anything, and if it is too small, the term won't have any meaningful effect.

The main differences between L_1 and L_2 regularization are:

- L_1 regularization is more likely to produce sparse weights, i.e. pushing some of the weights to zero. This could be useful for feature selection, as the model will learn which features are important and which are not (see image above).

- L_2 regularization is more likely to produce small weights, and spreads the penalty across all the weights.
- Therefore, and because L_2 is also easier to compute, it is more common in practice.

Weight decay is a similar term to the formula above:

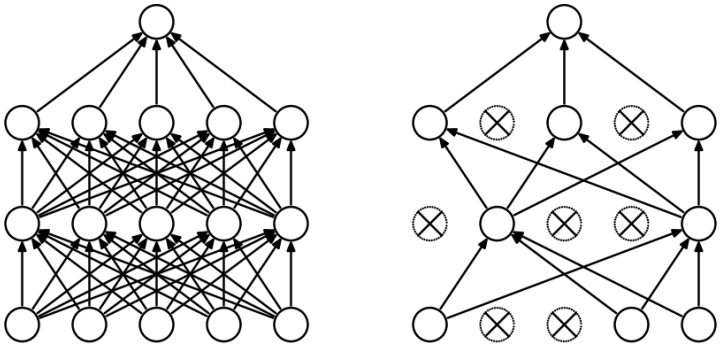
$$\theta_{t+1} = \theta_t - \alpha(\frac{\partial L}{\partial \theta_t} + \lambda\theta_i) = \theta_t - \alpha\frac{\partial L}{\partial \theta_t} - \alpha\lambda\theta_t$$

However,note the small difference:

- Weight decay is a term that is added to the optimizer, and not to the loss function. In regular SGD, weight decay is equivalent to the L_2 regularization term. The difference is in optimizers like Adam:

$$\theta_{t+1} = \theta_t - \alpha\left(\frac{\hat{m}_{t+1}}{\sqrt{\hat{v}_{t+1}+\epsilon}} + \lambda\theta_t\right)$$

Dropout



(a) Standard Neural Net (b) After applying dropout.

This is the most common and most useful regularization technique to date. The idea is to randomly "drop" some of the neurons in the network during training, so the model will not be able to rely on any single neuron, and will have to learn more general patterns. A key feature here is that it reduces the co-adaptation of neurons, and they will not be able to rely on each other.

Notes:

- Dropout introduces a hyperparameter p. In literature, it could be referred to as p_{drop} or p_{keep}, etc. In the original paper, it is the probability to **keep** a neuron, while PyTorch's implementation is the probability to **drop** a neuron.
- The dropout layer with p_{drop} will drop **exactly** p_{drop} of the neurons (or channels, for convs). For a single dropout layer, it also represents the probability of a neuron to be dropped.
- The dropout layer drops neurons only during training, and not during inference. At the latter stage, the neurons are scaled by p_{keep}, such that $x^\star = x \cdot p_{keep}$. This is done to keep the same magnitude (or expected value if you'd like) of the activations during inference, as this is what the next layer expects.
- At each iteration, the neurons to be dropped (or kept) are chosen randomly in respect to the parameter p, and therefore the model is trained on a different "subnetwork" at each iteration. This is why dropout is usually looked at as an **ensemble** of different models.
- Dropout is usually applied last in the computation block (Linear $\rightarrow$ Normalization $\rightarrow$ Activation $\rightarrow$ Dropout).

- The layer drops the neurons, not the weights.
- Inverse Dropout (Link): In order to save computations during inference, we could already scale the remaining neurons by the $\frac{1}{p_{keep}}$ parameter, and not during inference. An intuitive explanation would be: Assume that M is the value of the magnitude of the activations that are fed into the dropout layer, and assume a p_{keep}. During training, we drop $(1 - p_{keep})$ neurons, so the magnitude is already scaled down by p_{keep}. Therefore: $M \cdot p_{keep} \cdot \frac{1}{p_{keep}} = M$

```python
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  out = np.dot(W3, H2) + b3
```

test time is unchanged!

The Dropout layer operates differently for the Affine and Convolutional layers. For the Affine, single neurons are dropped randomly, while for the latter, entire channels are dropped. This is because due to the fact that convolutional layers capture spatial information, in contrast to the Affine layers, where features are learned by each neuron. Therefore, dropping channels retains the capability of extracting meaningful features, while still introducing regularization.

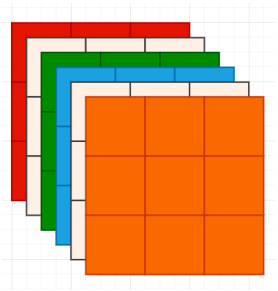


Figure IV.1: Spatial Dropout for convolutions with dropping probability $p_{\text{drop}} = \frac{1}{3}$. The brighter channels in this example image are dropped (All entries are set to zeros.

Data augmentation

A nice article: Link

Data augmentation is a technique that is used to artificially **virtually** increase the size of the training set, by applying random transformations to the data. That means, that the actual physical size of the training

set is not changed, rather at each epoch, the model is trained on different variations of the data, with some probability p. The idea is to make the model more robust to different variations in the data, and to prevent overfitting. The most common transformations are: rotation, scaling, translation, flipping, cropping, etc.

Notes:

- Data augmentation is usually applied to image data, but could be applied to any kind of data.
- The transformations are applied randomly, with some probability p.
- The transformation is done solely on the training set, and not on the validation or test sets.
- The transformations are usually applied in the data loader, and not in the model itself.
- Not all transformations are useful for all tasks. For example, rotating images of digits is not useful, as it will change the label of the image.
- Make sure that in supervised learning, the label of the data is also transformed accordingly, if needed.

IV.2.e Batch normalization

Batch Normalization (Link) is a technique that is used to normalize the neurons of the layer. The idea is to make the optimization process easier, as the activations will not be too large or too small, and the optimizer will be able to converge faster.

Note, that we normalize groups of neurons. Each group consists of **all the neurons in the batch** that are created by the same weights, either it's a set of weights in a FC layer or a kernel in a convolution.

For all corresponding neurons, perform the following normalization:

$$\hat{x} = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

Where μ is the mean of the activations, σ is the variance of the activations, and ϵ is a small number to avoid numerical instability. Then, scale and shift the activations:

$$y = \gamma\hat{x} + \beta$$

Where γ and β are learnable parameters, that are learned during training. The idea is that the model will learn the optimal mean and variance for the activations, and therefore the optimization process will be easier.

$$\begin{bmatrix}\gamma_1 & \gamma_2 & \dots & \gamma_M\end{bmatrix} \odot \begin{bmatrix} x_{1,1} & x_{1,2} & \dots & x_{1,M} \\ x_{2,1} & x_{2,2} & \dots & x_{2,M} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N,1} & x_{N,2} & \dots & x_{N,M} \end{bmatrix} + \begin{bmatrix}\beta_1 & \beta_2 & \dots & \beta_M\end{bmatrix}$$

Note: the mean μ and std σ are not necessarily 0 and 1, respectively.

For affine layers, the normalization is done across the batch for each neuron across the batch:

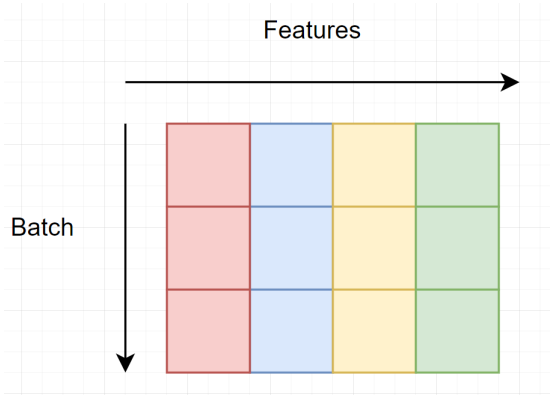


Figure IV.2: Batch normalization for affine layers. Mean and std are calculated for each color, across the batch.

- Note that this is the first and only layer that **breaks the abstraction** of the independence of the different instances in the batch, as the normalization of each neuron is done with respect to all the other corresponding neurons in the batch, i.e. for a neuron $v_{l,i,j}$, the normalization is done with respect to all the other neurons $v_{b,l,i,j}$, where $b \in \{1, 2, ..., B\}$.

However, for convolutional layers, the normalization is done across the batch, but not for each single neuron, but across the whole channels. Again, for all the neurons across the batch that **are created by the same kernel**:

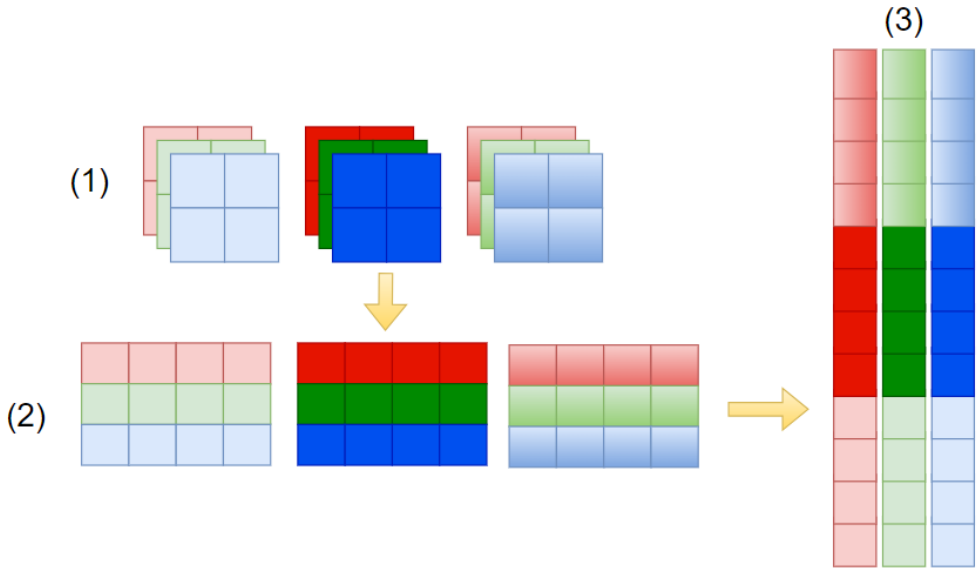


Figure IV.3: Batch normalization for convolutional layers. Mean and std are calculated for each channel, across the batch. We take all the corresponding channels across, and normalize them across all their entries. This could be seen as flattening them all and concatenating them, and then performing the normalization, just line in the affine layers.

- The normalization layers differ in their actions during training and inference. During training, the mean and variance are calculated for each batch, and the normalization is done with respect to them. However, during inference we have a single instance flowing through the network, so the statistics cannot be obtained in the same way. Therefore, during training we keep a running average of the mean and variance, and use them during inference:

$$\mu_{\text{running}} = \beta\mu_{\text{running}} + (1 - \beta)\mu_{\text{batch}}$$

$$\sigma_{\text{running}} = \beta\sigma_{\text{running}} + (1-\beta)\sigma_{\text{batch}}$$

Where β is a hyperparameter that controls the weight of the running average, usually set to 0.9. These running statistics are meant to approximate the mean and variance of their corresponding features or channels across the entire training set. During inference, we use those pre-computed statistics to normalize the activations.

- Also, the γ and β parameters are learned during training, and are used to scale and shift the activations. They are also used during inference. For affine layers, their shape is the same as the number of neurons in the layer, i.e. the dimension of each row in the activation matrix. For convolutional layers, their shape is the same as the number of channels in the layer tensor. Example:
  - For affine layers, assume a batch size of 8 and a layer with 64 neurons. Then, γ and β will be of shape (64,). The total amount of parameters is $2 \times 64 = 128$.
  - For convolutional layers, assume a batch size of 8 and a layer with 32 channels. Then, γ and β will be of shape (32,). The total amount of parameters is $2 \times 32 = 64$.

Problems with batch normalization:

- A small batch size: If the batch size is too small, the mean and variance of the activations might not be representative of the entire training set, and therefore the normalization will not be accurate, and can introduce too much noise. It has been shown (Link) that for a batch size smaller than 16, one should consider other normalization methods.
- Since the statistics are only approximations of the real ones over the whole training set, the batch norm is sometimes referred to as a "regularization" technique, as it introduces noise to the training process. However, this is not a good answer in the exam, and should not be a go-to solution if you're looking to solve overfitting.

Other normalization techniques: The following are alternative options to the default batch norm, and could be really useful in some cases. Note that they do not break the abstraction of the independence of the instances in the batch, and the normalization is done independently within each instance.

- Layer normalization: The normalization is done across the features - the entire instance. This was found really useful in transformers. In convolutional layers, it is the entire tensor.
- Instance normalization: Not really relevant for the vanilla FC layers. In convolutional layers, the normalization is done independently for each channel.
- Group normalization: The best alternative, and was shown to work better than BN for batch sizes smaller than 16. It is similar to layer normalization, but we divide the channels to groups, and normalize each group independently. Layer-norm could be seen as a special case of group-norm, where the number of groups is 1.

IV.2.f Hyperparameter tuning

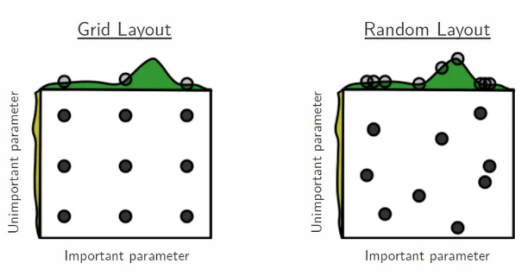
This stage is an important one in the deep learning pipeline. The idea is to find the optimal hyperparameters for the model, that will make it converge to a minimum, and generalize well on unseen data. The most common hyperparameters are, in a reasonable amount of time. Common hyperparameters are: learning rate, batch size, number of epochs, optimizer, size of hidden layers, activation functions, etc. However, while being crucial, it is quite easy to have a very good starting point, if we know our theory:

1. Learning rate - use a learning rate scheduler, to mitigate the problem of finding a bad initial one (A list of PyTorch built-in schedulers).
2. Activations - go with ReLU or its variants.

3. Optimizer - There are many amazing ones, but Adam and its variants are a great starting point (see: Link).
4. Size of hidden layers - always start small, and grow as you go.

That said, the theoretical step could be quite cumbersome, as the search space is usually very large (each hyperparameter adds a dimension), and the training process is very time-consuming. There are a few methods to tackle this problem:

- Grid search: The most basic method, where we define a grid of hyperparameters, and train the model on all possible combinations. This is very time-consuming, and not practical for large search spaces.
- Random search: A more advanced method, where we randomly sample the hyperparameters from a distribution and train the model on those. This is usually more efficient than grid search, as it doesn't require training on all possible combinations.
- Bayesian optimization: A more advanced method, where we explore (look on a wide range) and exploit (look on a narrow range) the search space. However, this method is usually more complex and requires more computational resources and doesn't outperform random search to a degree that justifies it.



IV.2.g Weight initialization

The learnable weights of a model are usually initialized randomly, and not to zero. If all the weights are initialized with the same value, they will not be able to learn complex patterns, as many groups of weights will be updated to the same value. Therefore, it is crucial that the weights are sampled randomly from some distribution. The range from which they are drawn must not be too big, as that could easily lead to exploding gradients through the backpropagation process. On the other hand, if the range is too small, the model will not be able to learn anything, as the gradients will be too small, and the optimization steps will be too negligible. Thus, we should follow some meaningful initialization schemes:

Xavier Initialization

Xavier Initialization, also known as Glorot Initialization. It aims to keep the scale of the gradients roughly the same in all layers, which is particularly useful for activation functions like sigmoid and Tanh.

This ensures the variance of the output is the same as the input $\rightarrow$ Gaussian with

- Mean: $\mathbb{E}[X] = 0$, $\mathbb{E}[W] = 0$
- Variance. Remember that $\mathbb{E}[X^2] = \text{Var}[X] + \mathbb{E}[X]^2$ and if X,Y are independent:

$$\text{Var}[XY] = \mathbb{E}[X^2Y^2] - \mathbb{E}[XY]^2$$

,

$$\mathbb{E}[XY] = \mathbb{E}[X]\mathbb{E}[Y]$$

Therefore,

$$\begin{aligned}\mathrm{Var}(s) &= \mathrm{Var}\left(\sum_i^n w_i x_i\right) = \sum_i^n \mathrm{Var}\left(w_i x_i\right) \\ &= \sum_i^n \left[\mathbb{E}\left(w_i\right)\right]^2 \mathrm{Var}\left(x_i\right) + \mathbb{E}\left[\left(x_i\right)\right]^2 \mathrm{Var}\left(w_i\right) + \mathrm{Var}\left(x_i\right) \mathrm{Var}\left(w_i\right) \\ &= \sum_i^n \mathrm{Var}\left(x_i\right) \mathrm{Var}\left(w_i\right) = n(\mathrm{Var}(w) \mathrm{Var}(x))\end{aligned}$$

- We want to ensure that the variance of the output $s = xw$ is the same as the input: $\mathrm{Var}(s) = \mathrm{Var}(x)$.

$$\mathrm{Var}\left(w\right) = n(\mathrm{Var}(w) \mathrm{Var}(x)) \rightarrow \mathrm{Var}(w) = \frac{1}{n}$$

- Note that n is the number of input neurons for the layer of weights you want to initialize. This n is not the number N of input data $X \in \mathbb{R}_{N \times D}$, but $n = D$.

For a layer with n_{in} input units and n_{out} output units, the weights W are initialized as follows:

- Using a normal distribution:

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

  Or (what we teach in class, and the version you should know for the exam):

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

- Using a uniform distribution:

$$W \sim \text{Uniform}\left(-\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}\right)$$

  Where n_{in} and n_{out} are the number of input and output units, respectively.

Kaiming Initialization

Kaiming Initialization, also known as He Initialization, is a variant of Xavier Initialization. It is specifically designed for layers with ReLU (Rectified Linear Unit) activation functions family. This method aims to address the problem of dead ReLU in deep neural networks by maintaining the variance of the activations across layers.

The ReLU activation function is defined as:

$$\text{ReLU}(x) = \max(0, x)$$

Because the ReLU function outputs zero for any negative input, the average output of neurons is not zero but positive, which affects the variance. To maintain the forward signal's variance, the initialization needs to take into account the rectification effect of ReLU. He et al. proposed scaling the variance of the weights by $\frac{2}{n_{\text{in}}}$, where n_{in} is the number of input units to the layer.

For a layer with n_{in} input units, the weights W are initialized as follows:

- Using a normal distribution:

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

- Using a uniform distribution:

$$W \sim \text{Uniform}\left(-\sqrt{\frac{6}{n_{\text{in}}}}, \sqrt{\frac{6}{n_{\text{in}}}}\right)$$

Summary

- **Xavier Initialization**:
  - Suitable for sigmoid and Tanh activations.
  - Considers both input and output layer sizes.
- **Kaiming Initialization**:
  - Tailored for ReLU activations.
  - Focuses on the size of the input layer only.

IV.2.h Optimization problems

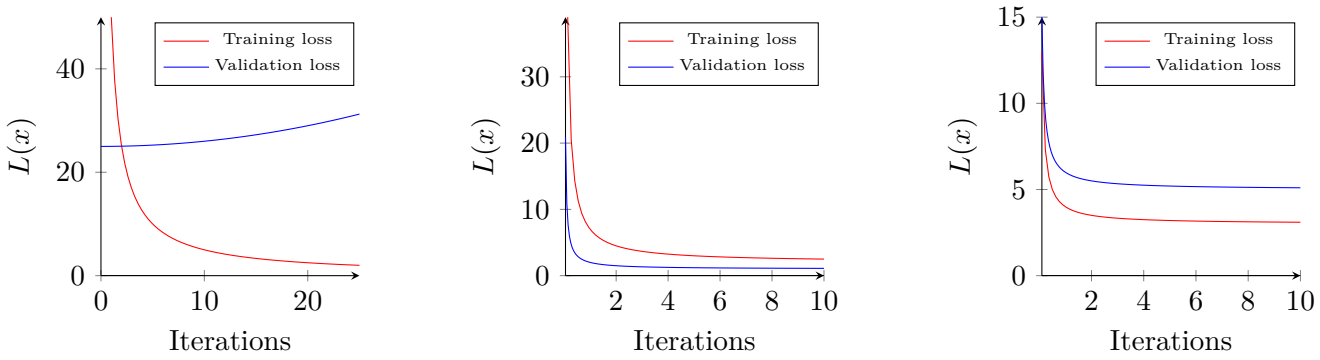
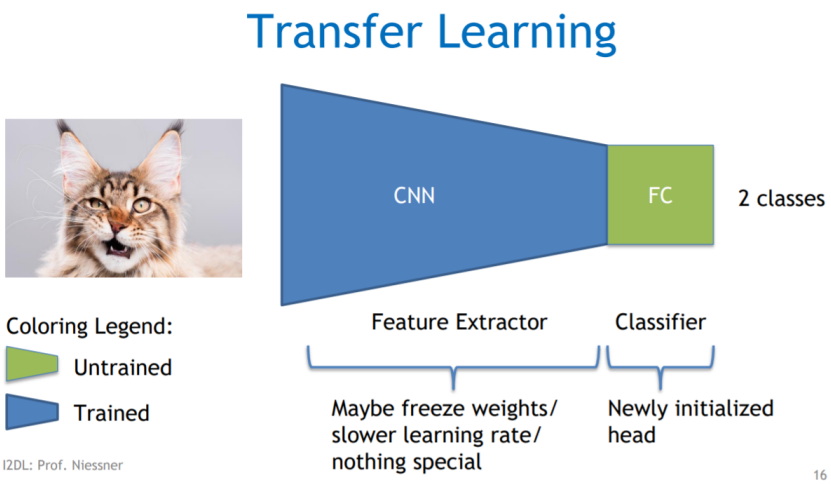


Figure IV.4: (a) - Mismatch of distribution, (b) - validation set is too easy / data leakage, (c) - Underfitting

- **Mismatch of distribution**: The training and validation sets are drawn from different distributions. The model learns the training set well, but fails to generalize to the validation set.
- **Validation set is too easy / data leakage**: The validation set is too easy, and the model is able to solve it with a very low loss. This could happen if the validation set is too small, or if the model is too simple and underfits the training set. Also, a bug in the implementation could cause data leakage, where some data from the validation set is used in the training process.
- **Underfitting**: The model is not able to solve the training set, and therefore the validation set as well. This could happen if the model is too simple, or if the optimization process is not good enough.

IV.3 Transfer Learning



Transfer learning is a machine learning technique that addresses the challenges of limited data and resource-intensive training. Instead of training a model from scratch, transfer learning utilizes a pre-trained model that has already learned features from a large dataset, and adapts it for a new but related task.

Key Concepts:

- **Pre-trained Models**: These models are trained on a large dataset (Distribution P1). They have learned useful features that can be transferred to other tasks.
- **Adaptation to New Task**: For a new task with a smaller dataset (Distribution P2), the pre-trained model is used as a starting point.
  - Only the final layer (the classifier) is replaced and trained on the new task's data.
  - Optionally, more layers can be fine-tuned with a lower learning rate if the new dataset is large enough.

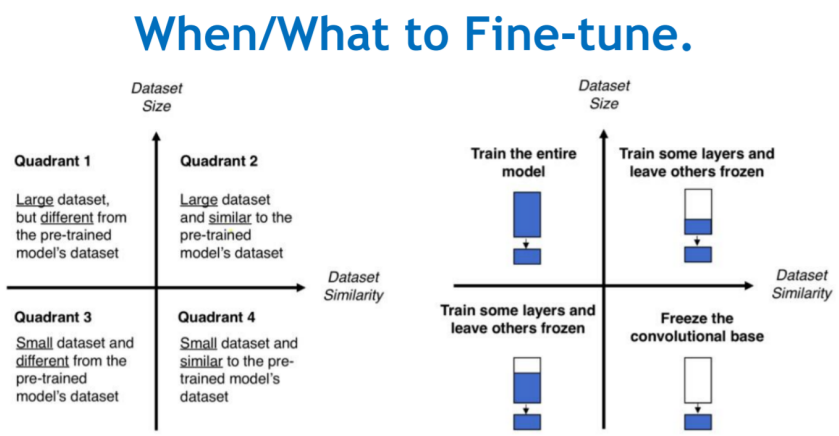
Benefits:

- **Efficiency**: Reduces the need for large amounts of labeled data and computational resources.
- **Performance**: Improves model accuracy by leveraging previously learned features.

When to Use Transfer Learning:

Transfer learning is effective when:

- The tasks share the same input type (e.g., both tasks use RGB images).
- The source task (T1) has significantly more data than the target task (T2).
- Low-level features from the source task are relevant to the target task.



IV.4 Questions

1. Optimizers:
   a) What is the main difference between gradient descent (GD) and stochastic gradient descent (SGD)?
   b) Name two advantages of SGD over GD.
   c) Why can we call RMS prop an adaptive optimizer?
   d) What is the bias correction in Adam? Why isn't it implemented in RMS prop?
   e) Adam with bias correction and Adam without bias correction have different global minimas. True or False?
   f) What two optimizers does Adam combine?
   g) Why is SGD+momentum usually better than SGD?
   h) i. Why do we always step in the direction of the negative gradient?
      ii. Write down a modified version of the SGD optimizer step, in case we want to maximize the loss instead of minimizing it.
2. Overfitting and underfitting
   a) Define overfitting in a short sentence.
   b) Define underfitting in a short sentence.
   c) State two different behaviors that indicate underfitting.
   d) State a possible reason for a situation where the validation loss is lower than the training loss.
   e) In a single word, what is the go-to solution to overfitting?
3. Regularization
   a) The term "regularization term" is ambiguous. What are the two usages of such "terms"?
   b) What is the effect of L_1 and L_2 regularization terms on the weights?
   c) What are the two differences between L_1 and L_2 regularization terms and "weight decay"?
   d) How can we avoid the computational overhead of the dropout layer during inference?

e) Explain how regular dropout works during training and during inference. Hint: crucial to distinguish between the definitions of p.

f) Why is data augmentation considered a regularization technique?

g) Why don't I allow you to consider Early-stopping as a regularization technique?

4. Batch Normalization:

a) Given a loss value L, and a fully connected layer with output shape 8×16, followed by a batch normalization layer: $y = BN(x) = \gamma * x_{norm} + \beta$

i. What are dimension of the parameters γ and β?

ii. Show a derivation of the gradients of those parameters $(\frac{\partial L}{\partial \gamma}, \frac{\partial L}{\partial \beta})$ and show how to use NumPy to calculate it.

b) Why is batch normalization sometimes referred to as a regularization technique?

c) Explain why we need to save the running averages of the mean and variance in the batch norm to the memory during training.

d) What optimization problems could batch norm help us solve?

5. Hyperparameter tuning

a) What is the main bottleneck for grid search?

6. Weight initialization

a) What weight initialization scheme fits the ReLU activation function? How does it affect the output of the activation function?

b) What do we expect to observe if we initialize the weights of the model to the same value?

7. Transfer Learning:

a) Explain one of the scenarios in the exercises where we used transfer learning, and how.

IV.5 Answers

1. Optimizers:

a) GD performs the optimization step at the end of the epoch, while SGD performs the optimization step at each iteration.

b) i. SGD performs more optimization steps in a single time unit and therefore converges faster.

ii. SGD introduces noise to the training process, and can help to avoid saddle-points.

c) RMS prop adapts the learning rate **for each parameter** individually, based on the magnitude of the gradients.

d) The bias correction is a mechanism to overcome the variables m, v's initialization bias to zero, so the early steps are really slow. It is not implemented in RMS prop, as it simply was first introduced with ADAM.

e) False. The global minimum isn't changed as the goal and loss don't change, but Adam with the bias correction would converge faster. However, they could end up in different local minims, as the first steps would be much bigger. For a good visualization of different minimas, look at the plots here.

f) RMS prop and Momentum.

g) SGD+momentum is usually better than SGD, as it helps the optimizer to escape saddle points and go faster in the appropriate directions.

h) i. Because the gradient points in the direction of the steepest increase (ascent) of the loss function, and we aim to get to the minima.

ii. Simply change the operation: $\theta_{t+1} = \theta_t + \alpha \frac{\partial L}{\partial \theta_t}$

2. Overfitting and underfitting

a) Overfitting is when the model performs too well on the training data, but fails to generalize to unseen data.

b) Underfitting is when the model fails to perform well on the training data.

c) The training loss is still decreasing when the model stops training / The accuracy of the model on the training set is very low.

d) The validation set is easier than the training set / underfitting stage at the beginning of training / bug in the implementation.

e) Regularization.

3. Regularization

a) i. A term that makes training difficult, hence forcing the network to generalize better (The go-to definition in the exam).

ii. A term that represents an additional objective to the training process.

b) The L_1 regularization term pushes some weights towards zero, thus making them sparser, while the L_2 regularization term spreads the penalty across all the weights, thus all become smaller, but not necessarily zero.

c) i. The L_1 and L_2 regularization terms are added to the loss function, while the weight decay is added to the optimizer step.

ii. In weight decay, the regularization term is also supervised by the learning rate, helping to mitigate the effect of its magnitude.

d) Use "inverse dropout", where already during training, we scale the remaining neurons by the $\frac{1}{p_{keep}}$ parameter, and not during inference.

e)
- During training: At each iteration, we drop neurons or whole channels with some probability p_{drop}.
- During inference: We scale all neurons by the p_{keep} parameter: $x^\star = x * p_{keep}$

f) Data augmentation is considered as a regularization technique, as it virtually increases the size of the training set, thus making the training process harder and therefore forces the model to learn more general patterns.

g) Early stopping doesn't make the training process harder, but rather stops it, when the generalization gap starts to grow. However, it will not solve the problem that the model is overfitting, but just doesn't let the current weights to get any worse. Saving checkpoints to the memory is a much better approach.

4. Batch Normalization:

a) i. Their dimension is 1×16.

ii. $y = BN(x) = \gamma * x_{\text{norm}} + \beta$

iii.

$$\frac{\partial L}{\partial \gamma} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial \gamma}$$

While $x_{\text{norm}} \in \mathbb{R}^{N \times D}$, both $\gamma, \beta \in \mathbb{R}^D$.

$$y = BN(x) = \gamma * x_{\text{norm}} + \beta = (1^N) \cdot \gamma * x_{\text{norm}} + (1^N) \cdot \beta \ \ //(\text{Broadcasting})$$

Where (1^N) is a column vector, but γ, β are row vectors. Therefore,

$$\frac{\partial L}{\partial \gamma} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial \gamma} = (1^N)^\top \cdot (\frac{\partial L}{\partial y} * x_{\text{norm}})$$

or $np.sum(dout * x_{\text{norm}}, axis = 0)$

$$\frac{\partial L}{\partial \beta} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial \beta} = (1^N)^\top \cdot \frac{\partial L}{\partial y}$$

or $np.sum(dout, axis = 0)$

b) Batch norm approximates the statistics of the whole training set on a small batch size, which introduces noise to the training process, thus making it a bit harder.

c) During inference (testing), the model is fed with a single instance at a time, and therefore there is no batch to calculate the statistics over. Therefore, we compute an approximation of the statistics during training, and use them during inference.

d) Batch norm normalizes neurons, therefore doesn't let them explode and get too small. It could mitigate the vanishing and exploding gradient problems.

5. Hyperparameter tuning

a) The main bottleneck for grid search is that the search space is usually very large, and the training process is very time-consuming. Therefore, grid search is not practical for large search spaces.

6. Weight initialization

a) The initialization scheme that fits the ReLU activation is the Kaiming or He initialization, that is a variant of Xavier initialization. With formula:

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

this initialization helps to maintain the variance of the activations across layers and prevents the dead ReLU problem.

b) If the weights of the model are initialized to the same value, the model will not be able to learn complex patterns, as many groups of weights will be updated to the same value. Therefore, the weights must be sampled randomly from some distribution.

7. Transfer Learning:

a) Autoencoders:

- Train in an unsupervised way an autoencoder to reconstruct the large dataset that is not labeled, to train an encoder that serves as a feature extractor. Then, remove the decoder and plug in instead a small classifier, that will train on the smaller labeled dataset, but with the capabilities learned on the bigger one.
- Train a segmentation network using a pretrained model as a backbone, either completely, or just the encoder, and do a smaller training session, to adjust the weights to the new dataset.

V Popular Architectures

Performance Metrics in ImageNet:

- **Top-1 Score**: check if a sample's top class is the same as its target label
- **Top-5 Score**: check if the label is in the first 5 predictions.
- **Top-5 Error**: percentage of test samples for which the correct class was not in the top 5 predicted classes.

V.1 LeNet

Firstly used for handwritten digits. Has the following architecture:

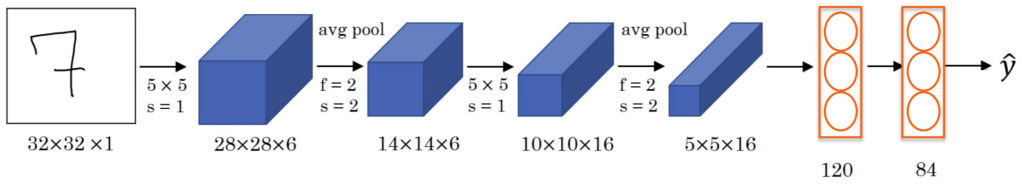


Figure V.1: LeNet Architecture: Conv → Pool → Conv → Pool → Conv → FC

- Apply valid convolution: size shrinks (reduced by two pixels on each side)
- $6 \times 1 \times 5 \times 5$ convolution filters used in first layer (6 filters, as the depth of the convolution obtained is 6)
- Average pooling is used (now: Max pooling much more common)
- Reduce first to 120, then to 84
- Tanh/sigmoid activation is used (not common now)
- Has 60k parameters
- As we go deeper: width, height go down and number of filters go up

V.2 AlexNet

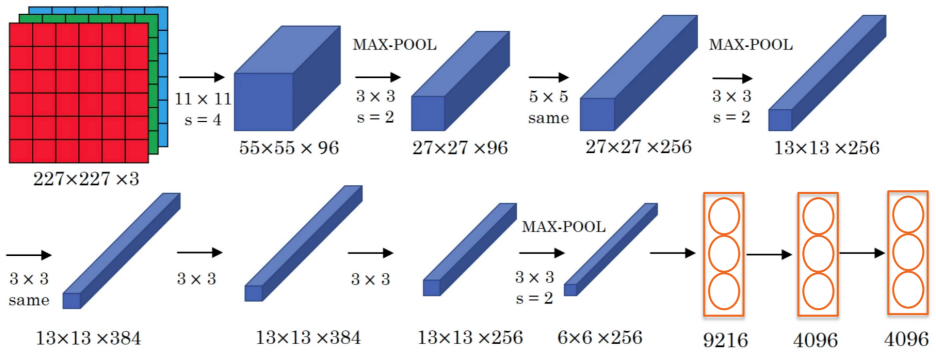


Figure V.2: AlexNet Architecture

- Similar to LeNet but ∼1000 times bigger
- Max Pool instead of Average (Non-linear)
- ReLU instead of Tanh/Sigmoid
- 60m parameters
- 8 layers
- **Note**: the first FC layer's dim (9216) is after the flattening operation of the tensor, and not an FC layer. The first FC layer is from 9216 → 4096.

V.3 VGGNet

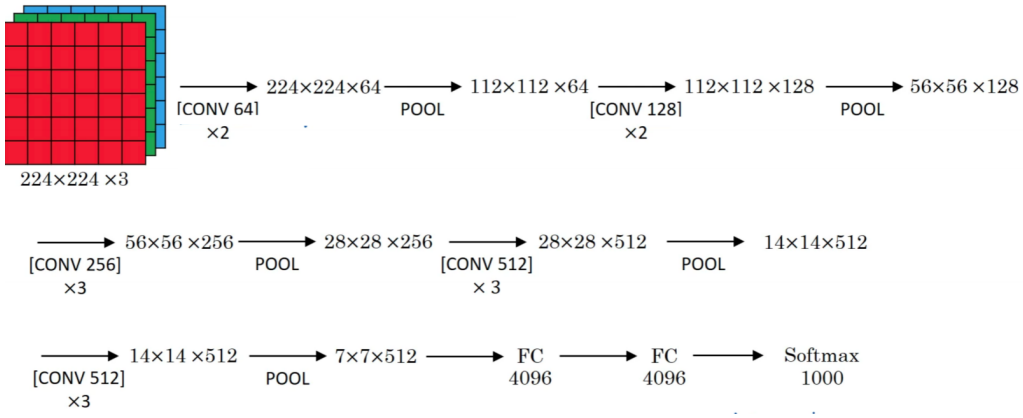


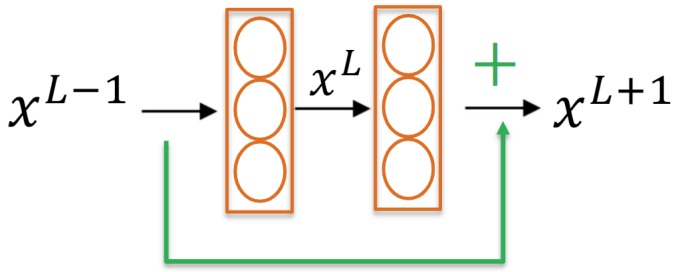
Figure V.3: VGGNet Architecture: Conv → Pool → Conv → Pool → Conv → FC

- CONV = $(k = 3, s = 1, p = 1)$
- Maxpool = 2x2 filters with stride 2
- 138m parameters
- It's large, but simplicity makes it appealing
- As we go deeper, width and height go down and number of filters up.

- 19 layers

V.4 Skip Connections: Residual Block

As neural networks had become deeper and deeper, harsh optimization problems, such as the vanishing gradients, were a big bottleneck in their capability to perform at full capacity. Skip connections, also known as residual connections, were introduced as a solution to this problem. They allow the gradient to flow directly through the network **("highway for gradients")** by providing alternate pathways for the gradient during backpropagation. This innovation not only mitigates the vanishing gradient issue, by allowing a full gradient to pass through, but also facilitates the training of much deeper networks. By enabling the network to learn identity mappings more easily, skip connections ensure that the performance of deep networks does not degrade with depth. Overall, skip connections have become a fundamental component in modern deep learning architectures, contributing to more stable and efficient training processes.



- Plain scheme $x^{L-1} \to x^L \to x^{L+1}$
- Main path: $x^{L+1} = f(W^{L+1}x^L + b^{L+1})$
- Add skip connection: $x^{L+1} = f\left(W^{L+1}x^L + b^{L+1} + x^{L-1}\right)$
- Highway for gradients: Assume that $Loss(x, \theta, y) = l$ and that $x^{L-1} \in \mathbb{R}^{N \times D}$, we show mathematically that a full magnitude of the gradients can pass from the loss function:

$$\frac{\partial L}{\partial x_{l-1}} = \frac{\partial L}{\partial x_{l+1}} \frac{\partial x_{l+1}}{\partial x_{l-1}} = \frac{\partial L}{\partial x_{l+1}} \frac{\partial (x_{l-1} + F(x_{l-1}))}{\partial x_{l-1}} = \frac{\partial L}{\partial x_{l+1}} \left(\frac{\partial x_{l-1}}{\partial x_{l-1}} + \frac{\partial F(x_{l-1})}{\partial x_{l-1}}\right) = \frac{\partial L}{\partial x_{l+1}} \left(1 + \frac{\partial F(x_{l-1})}{\partial x_{l-1}}\right)$$

- X^L is incorporated within $F(X^{L-1})$ and is irrelevant for this calculation.
- Note that x^{L-1} and x^{L+1} must have the same dimensions: If used by addition +, then the tensors' shapes must match completely. If concatenation along the channels, only the batch dimension and spatial size ($H \times W$) must match.
- If addition + is used, then we usually don't apply an activation on the second linear part of the residual block, but only after the addition.
- If addition + is used, the added number of calculations is negligible, to the point that we gain that much power for free. It's basically cheating.
- The identity is easy for the residual block to learn, so performance is guaranteed to not degrade in comparison to a plain network, w/o skip connections.

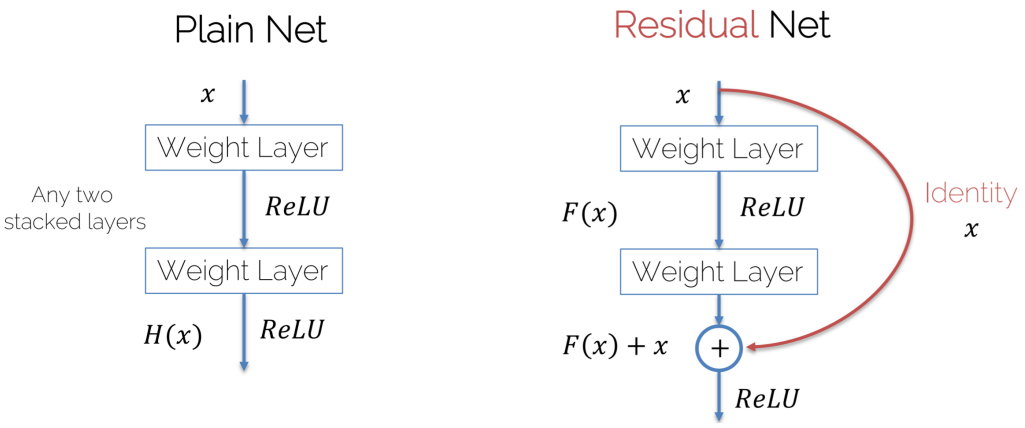


Figure V.4: Plain block vs. Residual Block

V.5 ResNet (Residual Networks)

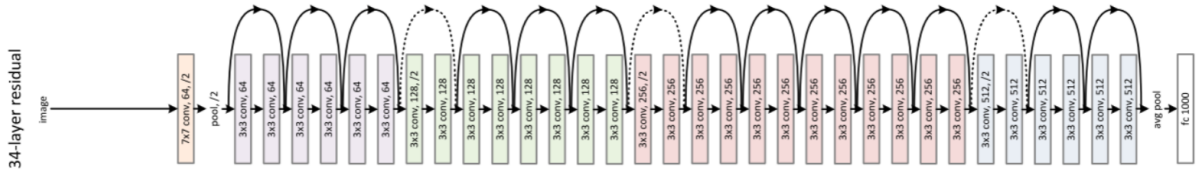


Figure V.5: ResNet

A deep network that utilizes the powerful skip connections and has become the go-to idea in any modern deep learning architecture.

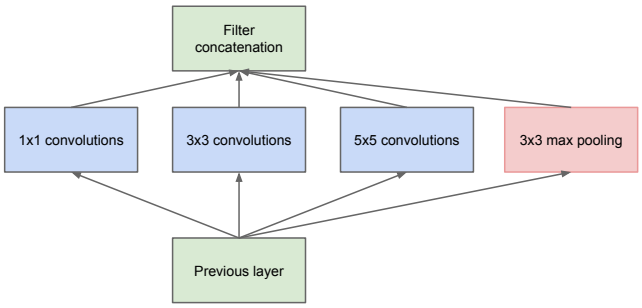
- 60m parameters
- 152 layers.
- Note that if there is dimensionality reduction, the gradients cannot flow completely uninterrupted all the way from the loss function to the very first layers, but only at blocks of the network where the dimensions are equal, where addition between two tensors is possible.

V.6 GoogleNet: Inception Layer

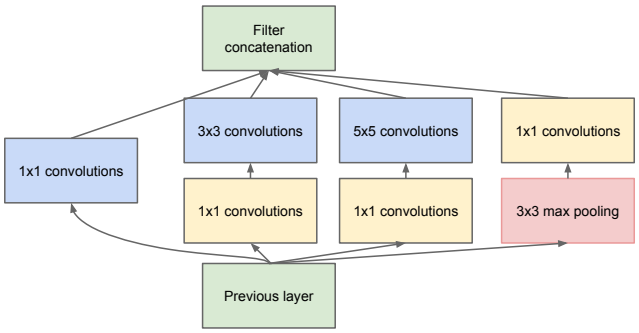
Why to restrict our model to a specific kernel size? We're Google, we have the computational power - we'll just use them all!

Inception layer:

- Each block in the network is made up with 4 different convolutions or pooling layers, that are then concatenated as an output.
- Very expensive to perform all of that. Therefore, 1×1 Convolutions are used, to reduce the number of channels and shrink the number of computations of the more expansive convolutions.
- Note that in order to maintain the same spatial sizes across all the different branches, the max-pool layer is initialized with the magic-trio: $k = 3, s = 1, p = 1$.

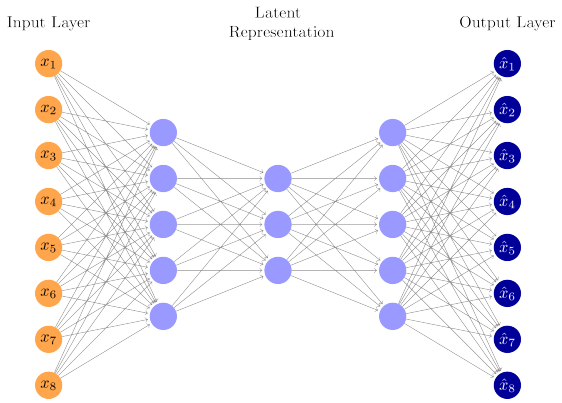


(a) Inception module, naïve version



(b) Inception module with dimensionality reduction

V.7 Autoencoder

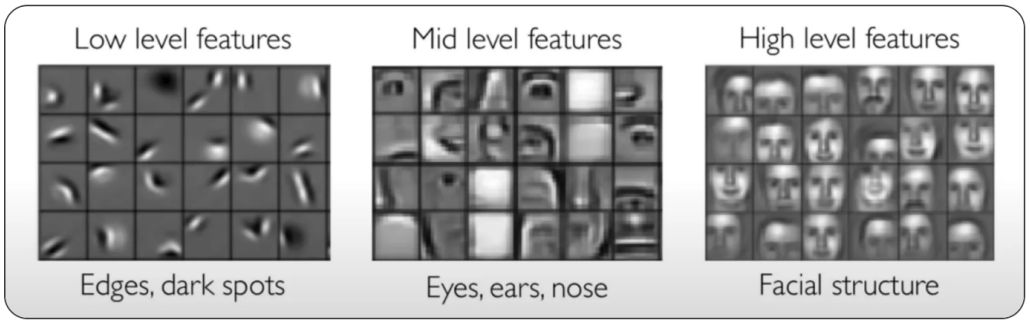


One of the most interesting network architectures. An autoencoder is such a powerful tool, that later on was perfected, and it is being used intensively in the industry and research, for the tasks of segmentation, 3D reconstruction, generative models, style transfers, etc. Basically, endless applications.

1. The basic architecture is based on fully connected layers.
2. It consists of three main parts:
   - Encoder: Reducing the dimensionality towards the **latent space**. That forces the network to focus on collecting the most meaningful features from the input data, so the loss of information would be as small as possible. Therefore, we could say that the Encoder is used as a **features extractor**.
   - Latent space: the compressed representation of the input data. These latent reside in multiple directionalities and have really powerful representations of features. By manipulating them correctly, one could achieve very interesting things.
     - If the size of is too small, not much information could eventually pass through to the decoder, and the reconstruction would be very hard. The result would be very blurry.
     - If too big, the network could basically learn to copy the image, without learning any meaningful features.
     - The latent space is a very powerful representation of the input. While RGB images, for example, offer quite redundant information on their own, the compressed latent space could represent the most meaningful features, that are specific for the task at hand, on a multidimensional manifold. From an RGB image, the latent space could represent so many things

and can even close the gap between two completely different domains. For example, one could align features from an RGB image with its corresponding semantic segmentation map, which differs from it completely, such that there is a simple linear transformation between them (Halperin et al. - may god help me, and soon it will be). One could learn "simple" features for reconstruction, or even the distribution of the dataset (see VAE later in this chapter).

– In the literature we say, that the latent space represents "high-level features", while the early layers of the model extract "low-level features":



– Calculations are usually **much** cheaper to perform on the latent space, which is a low-dim representation on the input. Therefore, we should aim to perform as little as possible calculations on higher-resolutions, and instead do most of the heavy-work on those lower-dim spaces, that are also much more flexible in what they can represent.

- Decoder: Receives the dimensionality-reduced latent space, and aims to **reconstruct** the input of the Encoder. The output has the same shape as the input.

3. Without non-linearities, it is very similar to PCA.

But why do we even want to use that? Auto encoders, as used in exercise 08, is an excellent solution to a state where our dataset is very big, but only just a small part of it is actually labeled, like medical a CT dataset, for example. So, we will have 2 steps:

1. Autoencoder → reconstruct the input. Let the Encoder learn the relevant features about the **unlabeled** data. This part can be referred to as **unsupervised learning**.
2. After the training has converged, remove the decoder and discard it. Then, plug in instead, just after the latent-space, a very simple fully-connected classifier, and train on the **labeled** data, given the fact that the remaining Encoder is already trained as a good features extractor. This part can be referred to as **supervised learning**.

V.8 Fully Convolutional Networks

Simply put, these are neural networks that consist of only of convolutional layers. Notes:

- with some restriction, these networks can accept different **spatial** sizes of inputs, with no any change to the architecture. Example: In exercise 9, we used input images with spatial sizes of 240×240, while the Alexnet (section V.2) backbone was originally taking inputs of shape 224×224. If FC layers are used, then the input size must be fixed, as they process the entire input at once. However, it is important to remember that convolutional layers are NOT invariant to changes in scale, and without any further training for adjustment to the new settings, the performance is likely to be suboptimal.

- In computer vision applications, these types of architectures are much superior to the original FC networks: they usually introduce much fewer parameters, fewer calculations and much more capable of extracting meaningful **local** features.

V.9 U-Net

A fully-convolutional autoencoder with skip connections.

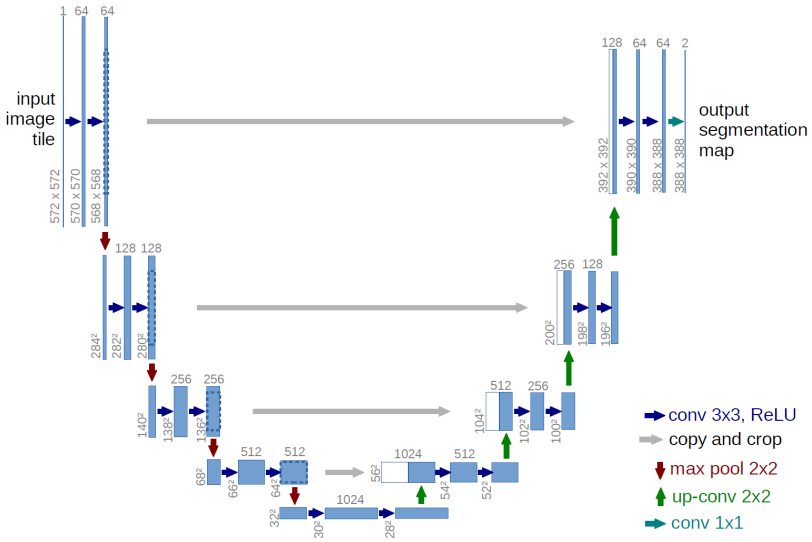


Figure V.6: A **generic** example of the U-net Architecture → could be implemented with much different hyperparameters.

- The U-net incorporates skip connections between corresponding layers in the encoder and the decoder, for the **highway of gradients**, and the usage of **fine-grained** features from the encoder, which performs as a **features extractor**. In contrast to the original ResNet (section V.5), the gradients flow to the shallower layers of the network with even fewer steps and could be even more meaningful than in a ResNet with dimensionality reduction, where gradients cannot flow completely up-the-stream.
- Remember that the main task of an autoencoder is to compress the features, for the selection of the most meaningful ones. However, as the spatial dimensions shrink, we face an issue of loss of critical information. To mitigate this, as we decrease the spatial size, we increase the number of channels, to offer more "breathing room" for features to pass through, and learn a bigger variety of them. But, when decoding the latent space, the spatial size increases, and therefore we must shrink the number of channels, to mitigate the exponentially rising number of calculations. A convolution on the original spatial size, no matter how thin it is parameters-wise, would introduce so many calculations and become a serious bottleneck for the runtime, while most processing could be done in a much cheaper fashion at the lower dimensions.
- The latent space here is a tensor of shape $B \times C \times H \times W$ and not a single vector, like in FC layers.
- These networks are widely used in tasks such as semantic segmentation, depth prediction, image reconstructions, GANs, etc.

V.10 Generative Networks

Very popular tasks, which their destructive results, that we encounter today all over, with fake-images flooding any good part of our lives. In this course, we introduce only two of the earlier ones: Variational Autoencoders (VAEs) and Generative Adversarial Networks (GANs). Stable-diffusion is for the advanced courses.

V.10.a Variational Autoencoders (VAEs)

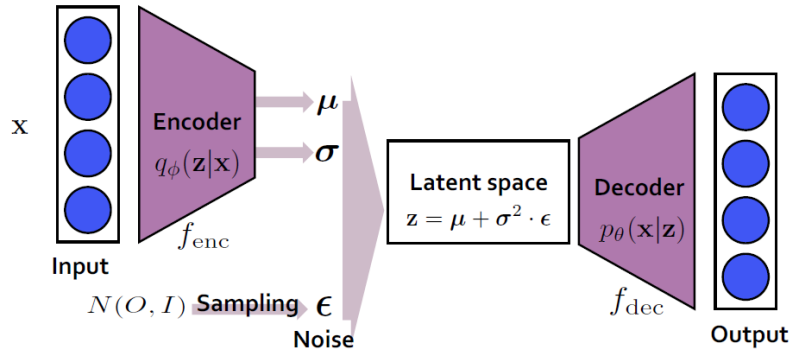


Figure V.7: During training: Encode the mean and std, use the parametrization trick to sample from the encoded distribution, and then decode the sample.

Based on the vanilla Fully-Connected Autoencoder (section V.7), instead of extracting features that will allow us to reconstruct the input, now we would like to encode the distribution of the entire training set into the latent space. This is done by introducing a probabilistic approach to the latent space, where the latent space is not a single vector, but two vectors (a single vector that is split in practice into two) that represent the compressed mean and the variance of the data's distribution. During training, we use both encoder and decoder to reconstruct the input. However, instead of using the encoded latent space, we sample from the distribution it represents (by some method that are beyond the scope of this course). We use the **parametrization trick** to sample from the actual encoded distribution:

$$z = \mu + \sigma\epsilon$$

where $\epsilon \sim \mathcal{N}(0, 1) \rightarrow$ sample noise from the standard normal distribution and then scale and shift the sample using the mean and variance of the encoded distribution.

The loss function is now composed of two parts:

1. The reconstruction loss: Just like in the vanilla Autoencoder, we would like to reconstruct the input as best as possible, to guide the encoder to learn the most meaningful features.
2. The KL-divergence loss: The KL-divergence loss is a measure of how much two probability distributions differ from each other. The loss function is defined as:

$$\mathcal{L} = \mathbb{E}_{p(z|x)}[\log p(x|z)] - \mathbb{E}_{q(z|x)}[\log q(z|x)] = \frac{1}{2}\left((\mu - \mu_{\text{target}})^2 + \log\left(\sigma^2\right) - \log\left(\sigma_{\text{target}}\right)^2 - 1\right)$$

   where μ and σ^2 are the parameters of the learned distribution, and μ^{target} and σ^{target} are the parameters of the target distribution, which is assumed to be the standard normal distribution, with mean 0 and variance (std) 1.

On the other hand, during testing (inference), we follow the following steps:

- Sample random Gaussian noise from the standard normal distribution.
- Use the **parameterization trick**.
- Decode the sample, and practically generate a new sample from the distribution of the training set.

Differences from regular autoencoders:

- The latent space is not a single vector, but two vectors that represent the mean and variance of the distribution of the training set.
- The loss function is composed of two parts: the reconstruction loss and the KL-divergence loss.
- During testing, we sample from the encoded distribution, and not from the encoded latent space.

V.10.b Generative Adversarial Networks (GANs)

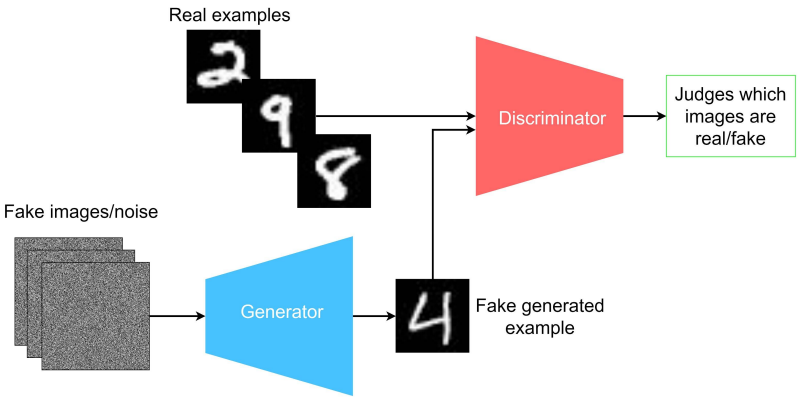


Figure V.8: During training: The generator generates fake images, and the discriminator tries to distinguish between real and fake images.

GeeksForGeeks has a great article on GANs: Link

GANs are a type of generative network that consists of two networks: a generator and a discriminator. The generator is responsible for generating fake images, starting from some random Gaussian noise, while the discriminator is responsible for distinguishing between real and fake images. The two networks are trained simultaneously and compete against each other, with the generator trying to generate images that are indistinguishable from real images, and the discriminator trying to distinguish between real and fake images. At each iteration of the training, we follow the following steps:

1. Sample random Gaussian noise from the standard normal distribution.
2. Generate a fake image and sample a random real image from the training set.
3. Push both the fake and the real images through the discriminator, while labeling the fake with class 0 and the real with class 1.
4. Update the weights of the discriminator.
5. Push again only the fake image into the discriminator, but this time with class 1 (trying to fool the discriminator).
6. Update the weights of the generator.

Unlike VAEs, GANs implicitly learns the distribution of the training set. In the original GAN, this is done by using the two competing loss functions:

- Decoder loss: Binary cross-entropy loss between the output of the discriminator and the target label (e.g. 1 for real images, 0 for fake images):

$$L_D = -\frac{1}{2N}\sum_{i=1}^{N}(y_i \log(D(x_i)) + (1-y_i)\log(1-D(G(z_i)))$$

  where $D(x_i)$ is the output of the discriminator for the i-th image, N is the batch size and y_i is the target label.

- Generator loss: since we update first the discriminator weights to try and push the fake images to the class 0, we take the compliment of the discriminator loss:

$$L_G = -\frac{1}{N}\sum_{i}^{N}\log(D(G(z_i)))$$

These two work in a MixMax fashion, which can cause quite often a case of underfitting, since both sides "pull the rope" or even modal-collapse, where the generator learns to generate only a single image, that is the most likely to fool the discriminator. This is why there are many variations of the GANs, such as the Wasserstein GAN, the Least Squares GAN, the Conditional GAN, etc.

However, GANs proved to generate more real-looking images than VAEs, but at a cost of training instability and complex training scheme. It was lately overthrown by other techniques, such as Stable Diffusion, etc.

V.11 Questions

1. Architectures:
   a) LeNet uses average pooling to reduce the spatial size. Give one advantage and one disadvantage of using average pooling over max pooling.
   b) In LeNet - what is the receptive field of a neuron in the first FC layer?
   c) Alexnet uses a 11×11 convolutional filter in the first layer. Name two disadvantages of using such a large filter.
   d) Alexnet uses ReLU instead of sigmoid or Tanh, as used in LeNet. Explain why it allows Alexnet to be deeper than LeNet, when coupled with the Kaiming initialization.
   e) VGGnet: What is the purpose of the convolutional part of the model? Why do we need the FC layers at the end?
   f) InceptionNet:
      i. What was the problem with the first version of InceptionNet? How was it solved?
      ii. We learned in class the MaxPool is usually used to reduce the spatial dimensions. Therefore, how was it possible to use it inside the Inception block, and concatenate its output to all other outputs?
2. Skip connections:
   a) Why can we say that skip connections introduce "highway of gradients"?
   b) Given a residual block $X_{l+1} = X_l + F(X_l)$, where $X_l, X_{l+1} \in \mathbb{R}$ - show the highway of gradients in the chain rule formula of $\frac{\partial L}{\partial X_l}$, given some loss value $L \in \mathbb{R}$
   c) Can you give a python-like implementation of the residual block?

3. AutoEncoders:
   a) Assume we use an autoencoder to reconstruct an image. What could be used as a loss function? What do we compare between? What kind of learning is it (supervised or unsupervised)?
   b) What is the effect of a latent space that is too small? What is the effect of a latent space that is too big?
   c) What linear approach does this kind of autoencoder resemble? What is the advantage of autoencoder over this method?
   d) State a scenario in which we would like to use an autoencoder for feature extraction.
4. U-net:
   a) Give 3 advantages of U-net over the vanilla Autoencoder
   b) How do we mitigate the drop in spatial size, so not to much information is lost?
   c) For the task image-reconstruction, does it make sense to use skip-connections between the encoder and the decoder? Explain.
5. Generative Networks:
   a) What is the difference between GANs and VAEs in the way they learn the distribution of the training set?
   b) In the Vanilla GAN, why is it prone to underfitting and mode collapse?
   c) In GAN, how do we train the generator to fool the discriminator?
   d) VAE: what are the two loss function we use, and what is their purpose?
   e) What do we assume the training set distribution to be in both VAEs and GANs?
   f) Sampling from a random distribution, that is not the normal distribution, is very hard. How does VAEs solve this problem?

V.12 Answers

1. Architectures:
   a) i. Advantage: all entries of the window get a gradient.
      ii. Disadvantage: It is a linear operation, while max pooling offers non-linearity.
   b) i. 11×11 is very expensive in both parameters and calculations.
      ii. It captures more global features than specific local features.
   c) The entire input image (32×32).
   d) It prevented the vanishing gradient problem, that is likely in deeper networks.
   e) i. The convolutional part is used as a features extractor.
      ii. The FC layers have global receptive field, which is very useful for the task of classification, for which the model is meant in the first place.
   f) InceptionNet:

i. The first version of InceptionNet is very complex, due to convolutions with large kernel sizes. The problem was solved by introducing 1×1 convolutions, to reduce the number of channels before the more expensive convolutions.

ii. Simply used the magical trio $k = 3, s = 1, p = 1$, that is used to keep the spatial dimensions.

2. Skip connections:

a) Skip connections allow skipping whole blocks of layers, by adding the input to the output. That allows the gradient of that input X to bypass the block, and not be diminished by it.

b)

$$\frac{\partial L}{\partial X_l} = \frac{\partial L}{\partial X_{l+1}} \frac{\partial X_{l+1}}{\partial X_l} = \frac{\partial L}{\partial X_{l+1}} \frac{\partial (X_l + F(X_l))}{\partial X_l} = \frac{\partial L}{\partial X_{l+1}} (\frac{\partial X_l}{\partial X_l} + \frac{\partial F(X_l)}{\partial X_l}) = \frac{\partial L}{\partial X_{l+1}} (1 + \frac{\partial F(X_l)}{\partial X_l})$$

c)

```
z = conv1(x)
z = relu(z)
z = conv2(z)
x = relu(x + z)
```

3. AutoEncoders:

a) Loss functions: MSE, MAE. We compare the input with the reconstructed image. It is an unsupervised learning approach.

b) If the latent space is too small, the reconstruction will be very hard and the result will be very blurry - underfitting. If the latent space is too big, the network could learn to copy the image, without learning any meaningful features, thus - overfitting.

c) PCA. The advantage of autoencoder over PCA is that it is non-linear, and can learn more complex features.

d) A scenario in which we would like to use an autoencoder for feature extraction is when we have a lot of unlabeled data, and we want to extract features from it. Then, we can use the pre-trained autoencoder for a supervised task.

4. U-net:

a) i. Skip connections

ii. Allows making most of the calculations and processing at lower dimensions, for a fraction of the cost.

iii. Accepts different input spatial sizes, with no any structural changes.

iv. Allows the extraction of meaningful local features instead of the global ones.

v. No, as the model can learn to simply memorize the image and send it to the output layer.

b) We mitigate it by increasing the number of channels, to offer more "breathing room" for features to pass through, and learn a bigger variety of them.

5. Generative Networks:

a) VAEs learn the distribution **explicitly** (directly), while the GANs learn the distribution **implicitly** (indirectly).

b) Both Generator and discriminator work against each other, so both could converge to suboptimal solutions. Also, the generator could learn to produce a single image, which is likely to fool the discriminator, and thus have a very good accuracy.

c) Send the fake image through the discriminator, but this time with class 1, like for the real image.

d) VAE: MAE/MSE for reconstruction, and the KL divergence for the latent space's distribution.

e) We assume the training set distribution to be the standard normal distribution.

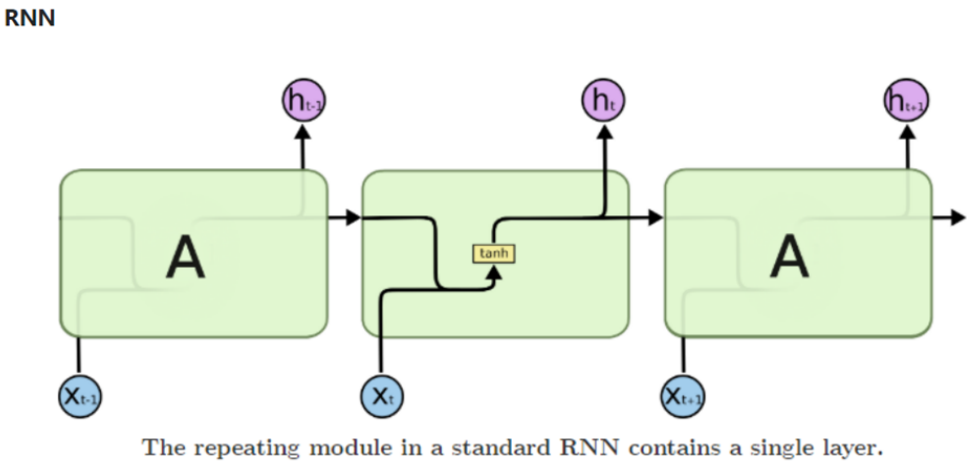
f) We use the parametrization trick: $z = \mu + \sigma\epsilon$, to shift the normal distribution to the actual learned distribution, where $\epsilon \sim \mathcal{N}(0, 1)$ is the sampled noise from the standard normal distribution, and μ and σ are the mean and variance of the distribution of the training set.

VI Recurrent Neural Networks and Transformers

So far we've been dealing with neural networks that took as input **independent** instances (e.g. single RGB images) and processed them for the sake of some task. In this section, we will be dealing with networks that process data instances that are dependent on each other with some context, e.g. sentences in text, videos - that are consecutive images of the same scene or even different parts of the same image, etc.

VI.1 Recurrent Neural Networks (RNNs)

The very first type of neural network that was designed to handle sequential data. The idea behind RNNs is to have a network that has some kind of memory of the previous inputs it has seen. This is achieved by having the network pass the output of the previous time step as an input to the current time step. This is illustrated in the figure below:



The repeating module in a standard RNN contains a single layer.

In this notion, we use **the same** weights to process each time step, but only in a very limited scope of the history, or "Short Term Memory". The components of the RNN are as follows:

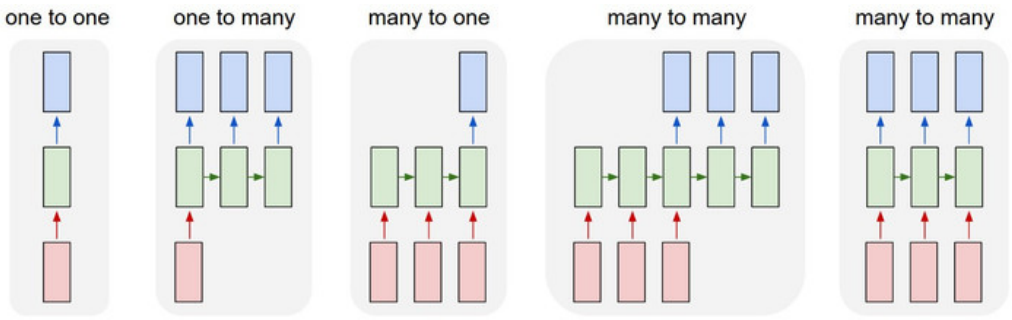
- x_t: The input at time t. Representing a word in a sentence, a frame in a video, etc. Usually given as a vector of a fixed size D, that in literature is called a "token". Note, that this scheme is used also in other context-related models, such as LSTMs and Transformers.
- h_t: The hidden state of time t. This is the memory of the network. It is a vector of size M. This hidden state is sent to the next time step and compresses into itself **all** previous time steps.
- o_t: The output at time t. This is the prediction of the network at time t. It is a vector of size M.

At each time step t, we use the same weights to process the current token x_t with the previous hidden state h_{t-1} to get the new hidden state h_t and the output o_t. x, h and o have a set of weights $W_x^{D\times M}$, $W_h^{M\times M}$ and $W_o^{M\times O}$ respectively. Note that these weights are shared across all time steps. Therefore, at time t we compute:

$$h_t = A(x_t \cdot W_x + h_{t-1} \cdot W_h + b_h)$$
$$o_t = A(h_t \cdot W_o + b_o), \quad o_t \in \mathbb{R}^M$$

Where A is an activation function (Sigmoid, Tanh or ReLU) and $b_\star$ is the relevant bias in the affine equation.

- Number of parameters: Without considering an output variable, the number of parameters in the RNN cell relies on the size of h, as the tensors need to match in shape for the addition. Assume $h \in \mathbb{R}^{1\times M}, x \in \mathbb{R}^{1\times D}$, then, the number of parameters is $M \cdot M + D \cdot M + M$, for W_h, W_x, b_h.



There are different types of RNNs, as shown in the figure above, and the outputs o_t could be used at each time step, only at the end of the sequence, or at every time step. The backpropagation through these networks is called "backpropagation through time" (BPTT) and is also affected by the choice of using the intermediate outputs, of course. This process heavily relies on the length of the input sequence and therefore could be quite expensive and long to compute.

Uses of the different types of RNN (source):

1. One-to-one: A simple neural network, as we know it.
2. One-to-many: Image captioning.
3. Many-to-one: Sentiment analysis.
4. Many-to-many (first sequence, then output) - Language translation, as the entire sentence is processed first.
5. Many-to-many (both input and output): Labeling images in a video, or some task that doesn't rely completely on the entirety of the sequence.

Also, the way h_0 is initialized can affect the performance of the RNN. Common initializations are either to a set of zeros, or sampled from a random distribution, but could also be learned.

Main challenges:

- Backpropagation through time (Example). Assume an RNN with 2 time steps, and a single output at the end, for simplicity. Also, for the sake of a readable example, assume all variables are scalars ($x, h, W_x, W_h, W_o, b_h, b_o \in \mathbb{R}$) and that the activation function is the identity function. Remember that if $y = f(x)g(x)$, then

$$\frac{\partial y}{\partial x} = f'(x)g(x) + f(x)g'(x) \rightarrow \frac{\partial y}{\partial x} = \frac{\partial f(x)g(x)}{\partial x} = \frac{\partial f(x)}{\partial x}g(x) + f(x)\frac{\partial g(x)}{\partial x}$$

  . The forward pass is as follows:

  - $y_1 = x_1 \cdot W_x + h_0 \cdot W_h + b_h$
  - $h_1 = A_1(y_1)$
  - $y_2 = x_2 \cdot W_x + h_1 \cdot W_h + b_h$
  - $h_2 = A_2(y_2)$
  - $y_3 = h_2 \cdot W_h + b_o$

– $o = A_3(y_3)$

Now, let's calculate the gradient of $\frac{\partial o}{\partial W_h} = \frac{\partial A_3(y_3)}{\partial W_h}$:

$$\frac{\partial A_3(y_3)}{\partial W_h} = \frac{\partial A_3(y_3)}{\partial y_3} \cdot \frac{\partial y_3}{\partial W_h} = \frac{\partial A_3}{\partial y_3} \cdot (W_h \cdot \frac{\partial h_2}{\partial W_h} + h_2)$$

$$\frac{\partial h_2}{\partial W_h} = \frac{\partial A_2(y_2)}{\partial W_h} = \frac{\partial A_2}{\partial y_2} \cdot \frac{\partial y_2}{\partial W_h} = \frac{\partial A_2}{\partial y_2}(W_h \cdot \frac{\partial h_1}{\partial W_h} + h_1)$$

$$\frac{\partial h_1}{\partial W_h} = \frac{\partial A_1(y_1)}{\partial W_h} = \frac{\partial A_1}{\partial y_1} \cdot \frac{\partial y_1}{\partial W_h} = \frac{\partial A_1}{\partial y_1} \cdot h_0$$

$$\frac{\partial A_3(y_3)}{\partial W_h} = \frac{\partial A_3(y_3)}{\partial y_3} \cdot \frac{\partial y_3}{\partial W_h} = \frac{\partial A_3}{\partial y_3} \cdot (W_h \cdot \frac{\partial A_2}{\partial y_2}(W_h \cdot (\frac{\partial A_1}{\partial y_1} \cdot h_0) + h_1) + h_2)$$

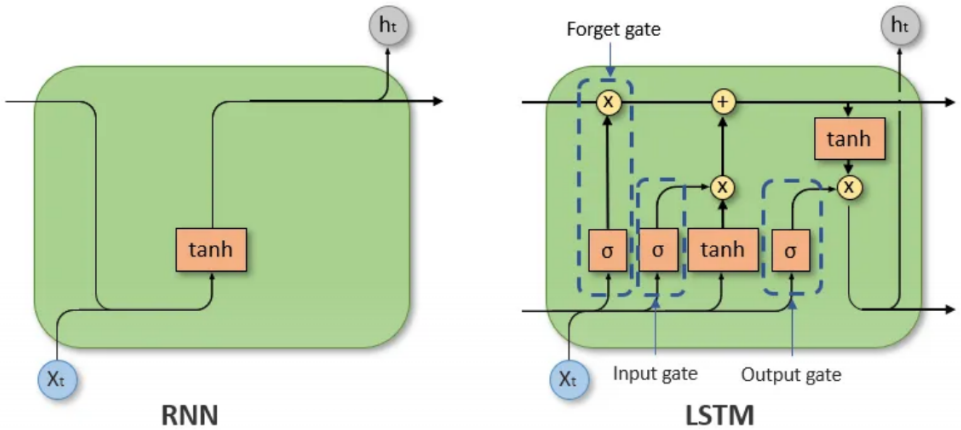
Where,

$$\frac{\partial A_3}{\partial y_3}, \frac{\partial A_2}{\partial y_2}, \frac{\partial A_1}{\partial y_1} = 1$$

- Observe the example above. We could see that the weight matrix is multiplied by itself as we backpropagate through time. If the eigenvalues of the weight matrix are smaller than 1, then the gradient will become smaller and smaller as we go up the stream - and introduce the vanishing gradient problem. On the other hand, if the eigenvalues of the weight matrix are larger than 1, then the gradients will explode! A solution could be to use a regularization term to force the weight matrix to be orthogonal, or clip the gradient values, so it won't get too small or too large.
- Not much capacity: While it is very cheap in parameters, using only 3 different weight matrices does not introduce much capacity to the network.
- Can only have short-term context, which makes context-related tasks, such as language translation, quite hard.

VI.2 Long Short Term Memory (LSTM)

LSTMs try to overcome the vanilla RNN's problems by changing the inner logic of the RNN cell by using different gates for past and current information and introducing skip-connections.



VI.3 LSTM Gates Explanation

The LSTM (Long Short-Term Memory) network is a type of RNN that addresses the vanishing gradient problem and can capture long-term dependencies more effectively. The diagram provided illustrates the internal workings of an LSTM cell. Let's break down the key components and gates of the LSTM block:

VI.3.a Components of LSTM

- **Cell State (C_t)**: The cell state runs straight down the entire chain, with only some minor linear interactions. This allows information to flow unchanged and provides a memory that can be updated or reset based on the gates' actions. Must have the same shape as the hidden state h.
- **Hidden State (h_t)**: This is the output of the LSTM cell at each time step, which can also serve as an input to the next time step.

VI.3.b Gates in LSTM

LSTM networks have three main gates that regulate the flow of information:

- **Forget Gate (f_t)**:
$$f_t = \sigma(h_{t-1} \cdot W_{f_h} + x_t \cdot W_{f_x} + b_f)$$
The forget gate decides what information to discard from the cell state. It takes the previous hidden state (h_{t-1}) and the current input (x_t) and passes them through a sigmoid function (σ). The output is a value between 0 and 1 for each number in the cell state C_{t-1}, where 1 means "completely keep this" and 0 means "completely forget this."
- **Input Gate (i_t)**:
$$i_t = \sigma(h_{t-1} \cdot W_{i_h} + x_t \cdot W_{i_x} + b_i)$$
The input gate controls how much of the new information (candidate values), which is generated by the Tanh activation, to add to the cell state. Similar to the forget gate, it uses a sigmoid function to decide which values to update.
- **Candidate Layer ($\tilde{C}_t$)**:
$$\tilde{Can}_t = \text{Tanh}(h_{t-1} \cdot W_{C_h} + x_t \cdot W_{C_x} + b_C)$$
This layer generates new candidate values, which could be added to the cell state. The Tanh function outputs values between -1 and 1.
- **Output Gate (o_t)**:
$$o_t = \sigma(h_{t-1} \cdot W_{o_h} + x_t \cdot W_{o_x} + b_o)$$
The output gate determines what the next hidden state (h_t) should be. It takes the input and previous hidden state, processes them through a sigmoid function, and then multiplies the output with the Tanh of the updated cell state to generate the hidden state.

VI.3.c LSTM Cell Update Equations

The cell state and hidden state are updated as follows ($\odot$ for element-wise multiplication):

- **Update the Cell State (C_t)**:

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{Can}_t$$

  The cell state is updated by multiplying the previous cell state by the forget gate value and adding the product of the input gate value and the candidate values.

- **Compute the Hidden State (h_t)**:

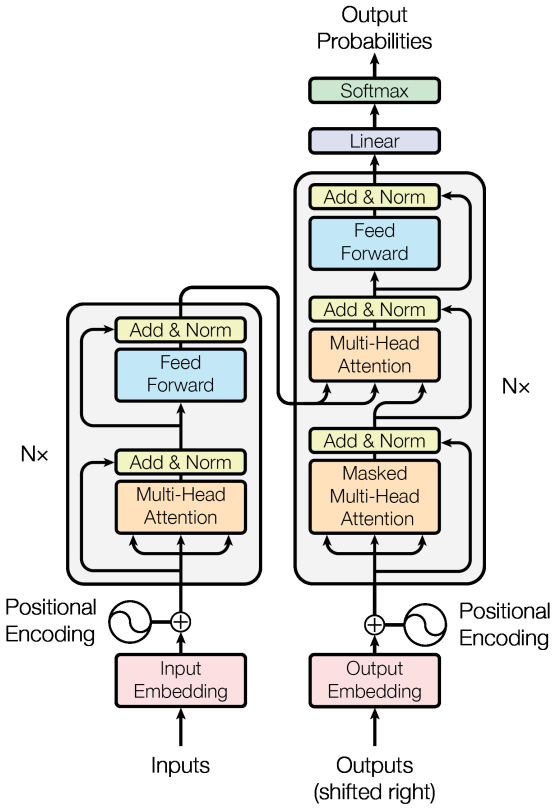
$$h_t = o_t \odot \text{Tanh}(C_t)$$

  The hidden state is updated by multiplying the output gate value with the Tanh of the new cell state.

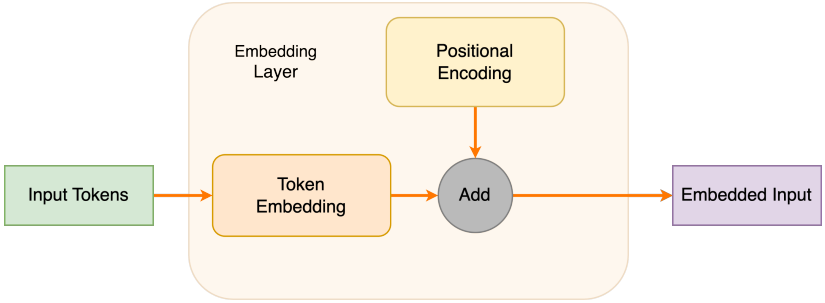
VI.4 Transformers: Revolutionizing Neural Network Architectures

Transformers have become a cornerstone in modern deep learning, particularly in natural language processing (NLP). Introduced by Vaswani et al. in 2017 (paper), Transformers have surpassed traditional recurrent neural networks (RNNs) and convolutional neural networks (CNNs) in various tasks due to their ability to handle long-range dependencies and parallelize training.

The Transformer architecture relies heavily on self-attention mechanisms, eliminating the need for recurrence, in contrast to RNNs and LSTMs. It is made up of an encoder-decoder structure, where both the encoder and decoder are composed of a stack of identical blocks. Each block consists of sub-layers, including multi-head self-attention mechanisms, feed-forward neural networks, and normalization layers.



VI.4.a Embedding Layer



Token Embedding

The sequence input to the Transformer architecture is made up of "tokens", which are fixed-size vectors, that are predefined, usually by some preprocessed "dictionary". The transformer cannot accept tokens of different sizes, as it is crucial for the mathematical operations within the different components. The embedding layer converts input tokens into dense vectors that represent the tokens in a high-dimensional space.

Positional Encoding

Transformers lack the sequential nature of RNNs, so they require a way to incorporate the order of the sequence. This is done through positional embeddings. The first proposal was the sinusoidal positional encoding:

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

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

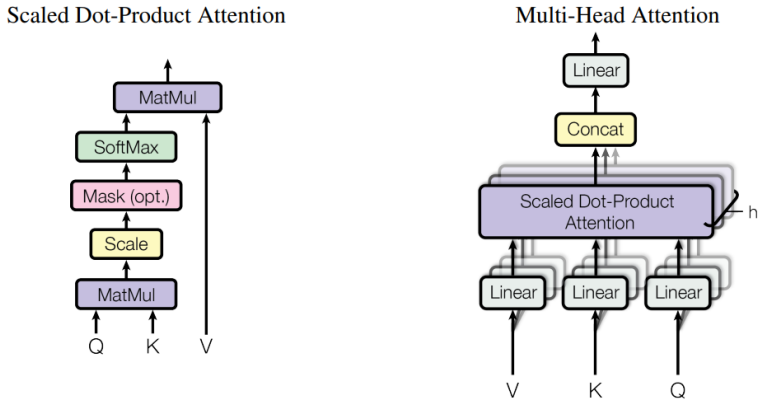
These functions provide unique encodings for each position, allowing the model to differentiate between different positions in the sequence. Before the input embeddings are fed into the encoder, those positional embeddings are added to their corresponding input tokens.

For a deeper understanding of embedding layers, take a look at this 3B1B video.

VI.4.b Attention Mechanisms

The self-attention mechanism is at the heart of the Transformer model. It allows the model to focus on different parts of the input sequence when encoding a particular word.

Scaled Dot-Product Attention



The scaled dot-product attention computes the attention scores using the following formula:

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

where Q (query), K (key), and V (value) are the input matrices, and d_k is the dimension of the key vectors.

- The division by $\sqrt{d_k}$ is another mechanism to prevent the outputs of the linear operations from becoming too huge, and in turn make the outputs and the gradients explode. In programming, you would usually see your loss value become *nan*.
- The Softmax function has two purposes - to normalize the weights to be in the range of [0, 1], and to make stronger weights stronger, and smaller weights - smaller

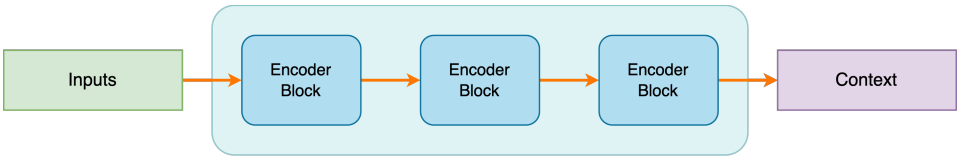
Multi-Head Attention

Multi-head attention allows the model to jointly attend to information from different representation sub-spaces. We simply divide each token into sub-tokens named "heads", and perform the linear operations with smaller weight matrices, but each head has its own set. Instead of performing a single attention function, the transformer employs multiple attention heads to capture different aspects of the input data. Each head performs its own attention calculation, and the results are concatenated and linearly transformed.

Simply reshape your tensor of shape (N, C, D) to $(N, C, D//\text{head-size}, \text{head-size})$ before feeding it to the self-attention mechanism. After the attention, the "concatenation" is performed by simply reshaping the tensor back to (N, C, D).

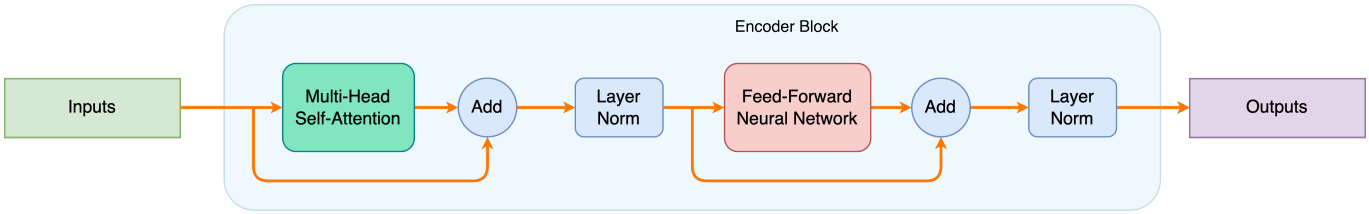
VI.4.c Encoder

The encoder is a stack of N identical blocks:



Each block consists of two main sub-layers:

- **Self-Attention Mechanism:** This allows the model to weigh the importance of different tokens in a sequence, in respect to each other.
- **Feed-Forward Neural Network:** A position-wise fully connected feed-forward network applied independently to each position.



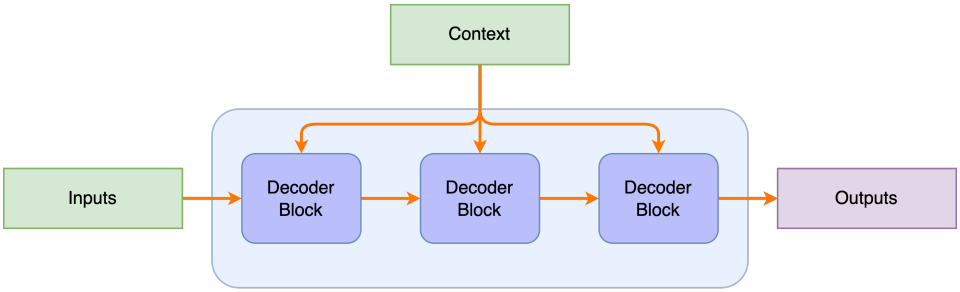
Each sub-layer has a residual connection around it followed by layer normalization. The output of each sub-layer is:

$$\text{LayerNorm}(x + \text{Sublayer}(x))$$

Note that those normalizations are crucial, as the self-attention mechanisms do not introduce any non-linear activations or normalizations, and therefore are prone to exploding values or gradients.

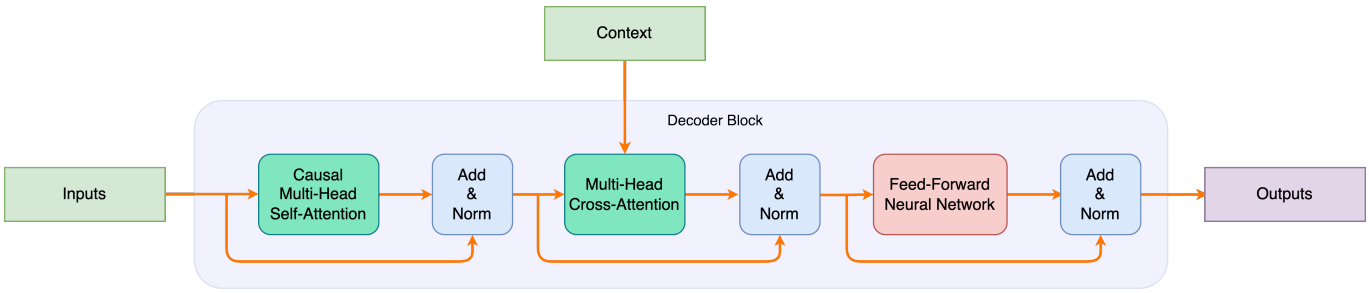
VI.4.d Decoder

The decoder is also composed of N identical blocks:



With an additional sub-layer compared to the encoder, each layer consists of:

- **Masked Self-Attention Mechanism:** Prevents attending to future tokens. This ensures that the predictions for the i-th position can depend only on the known outputs at positions less than i. Prevents from the decoder to simply copy its inputs.
- **Encoder-Decoder Attention:** Attends to the encoder's output. Let M be the embedding size - q is the **decoder**'s masked self-attention outputs, of shape $q \in R^{K \times M}$. k and v are the encoder's outputs, of shape $N \times M$. That results in K embeddings, to be processed and compared with the K ground truth embeddings.
- The output of the network, after applying N-decoder blocks, is a 2D matrix for each instance in the batch (each sequence) of shape $N \times C$, where N is the number of tokens in the sequence, and C is the number of classes in the **target** dictionary, for a classification loss function (CE).



VI.5 Questions

1. Give two drawbacks of using RNNs that are not the exploding or vanishing gradients.
2. How did LSTM solve the vanishing gradient problem in RNNs?
3. Is it a good idea to use ReLU instead of Sigmoid as the activation of the input gate of LSTM?
4. Transformers:
   a) Can the transformer architecture take embeddings of different sizes?
   b) Can it take sequences of different sizes as inputs to the encoder and the decoder?
   c) Cross-attention layer. Given the encoder outputs of shape $X_e \in \mathbb{R}^{N \times M}$ and the decoder outputs of shape $X_d \in \mathbf{R}^{K \times M}$, what is the dimension of the output of that layer?
   d) Is the task of predicting the next token in the sequence a classification or a regression task?
   e) Why is the self-attention mechanism prone to exploding gradients? How does the original architecture of the transformer solve that?
   f) Why is it important in transformers to use positional encoding, in comparison to RNNs?

VI.6 Answers

1. a) Too few parameters - not enough capacity.
   b) Short-term memory - cannot learn relevant context with tokens that are far away in the sequence.
2. By introducing the cell's state - a highway for gradients, just like with skip-connections.
3. No, it is not recommended. The sigmoid is bounded between [0, 1], and therefore allows us to choose what information to pass on, and with what magnitude. The ReLU function, however, doesn't have an upper bound, which drops this desired behavior of the sigmoid and could even cause numerical instabilities.
4. Transformers:
   a) No, the fully-connected layers within the transformer architecture expects a fixed size embedding input.
   b) Yes, sequences of embeddings are like batches of tokens, and the fully-connected layers know to handle different sizes of batches. In addition, the cross-attention in the decoder handles different sequence sizes properly.

c) $q = X_d, k = X_e, v = X_e$.

$$out = (q \cdot k^\top) \cdot v \to (K \times M) \cdot (M \times N) \cdot (N \times M) = (K \times N) \cdot (N \times M) = K \times M$$

d) Classification. Each word in the dictionary has its own label.

e) The self-attention model doesn't include any normalization layer, besides the $\sqrt{d_k}$ scale. The affine layers then could lead to huge numbers and therefore huge gradients. The transformer then utilizes layer norms, to normalizes each embedding individually before and after the attention blocks.

f) Transformers do not process the sequence input sequentially. Since the scores between two embeddings is invariant to permutations in the order of the tokens, unlike RNNs, we need to introduce the positional encoding to introduce time-wise context.

VII Appendix

VII.1 Multidimensional derivatives

VII.1.a Affine layer

$$y = XW + b \tag{VII.1}$$

Where $X_{NxD} \quad W_{DxM} \quad b_{1xM}$.

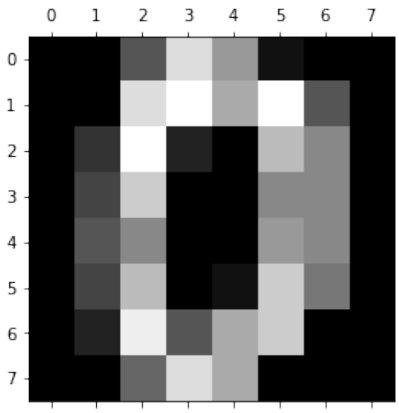
A known use case is the 1-dim case of a line equation:

$$y = ax + b$$

What is X?

- In the affine layer context, the matrix X is considered to be the input.
- In neural networks, we almost always refer to it as a **batch** of input elements (e.g. images).
- In some deep learning applications (e.g. "style-transfer"), it is also trained by backpropagation.
- Besides being the input of each layer of the network, X is also the **output** of the previous layer.

Let us take for example this one input instance (image) from the **MNIST** handwritten digits' dataset. Each **gray scale** image in this dataset is a $1 \times 8 \times 8$ tensor: 1 for the channels, 8 for the height, and 8 for the width.



For the affine layer, as phrased in (VII.1), each input instance is **flattened** to be a row vector inside X. Let us take a batch of 2 images from the MNIST dataset.

$$X = \left[\begin{bmatrix} x_{111} & \dots & x_{118} \\ \vdots & \ddots & \vdots \\ x_{181} & \dots & x_{188} \end{bmatrix} \begin{bmatrix} x_{211} & \dots & x_{218} \\ \vdots & \ddots & \vdots \\ x_{281} & \dots & x_{288} \end{bmatrix}\right] \rightarrow \left[\begin{bmatrix} x_{111}, & \dots & , x_{118}, & , x_{121}, & \dots & , x_{181}, & \dots & , x_{188} \\ x_{211}, & \dots & , x_{218}, & , x_{221}, & \dots & , x_{281}, & \dots & , x_{288} \end{bmatrix}\right] \tag{VII.2}$$

Here, the batch shape is $2 \times 1 \times 8 \times 8$

Question: What if we had a 3-channels RGB images?

Answer: The images are flattened the same, row by row and channel by channel. The actual order doesn't matter, but it is important that it will remain consistent among all input instances, so the weights will correspond to the correct entries.

What is W?

- The coefficient matrix.
- In a learning model, they represent the **learnable weights**, and modified during the backpropagation step.
- If in X each row represents one input inside the batch, in W each column represents the weights that are attached from all input neurons (cells in the input vector) to one neuron in the next layer, which is the input to the next layer, as could be seen in Figure VII.1.

Notes

- **Note**: It is not a linear function, but we treat it as an approximation. Why not? It doesn't follow the rules of linearity, where
$$f(x+y) = f(x) + f(y)$$
or
$$f(ax) = af(x)$$
- Another common notation of an affine layer is
$$y = Wx + b = \Big((Wx+b)^T\Big)^T = (X^T W^T + b^T)^T \tag{VII.3}$$
Which calculates the exact same thing, but results in a column vector and not a row. W weight vectors are now row vectors and X inputs are now column vectors. It is just a matter of how we construct our inputs and weights.

VII.1.b Derivatives

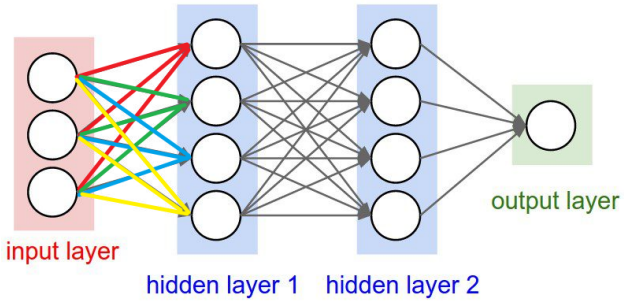


Figure VII.1: A neural network computational graph. Note: Although we always deal with batches of inputs, in the sketch, the input layer represents only one input instance (e.g one flattened image). Each color represents a different weights column vector in W. Also, each neuron in the input layer (true to any neuron in the network) will collect the gradients from the flow on the colorful edges that are attached to it.

What is a gradient

It all depends on the function!

- $$f : \mathbb{R} \to \mathbb{R}, \quad x \in \mathbb{R}, \quad \frac{\partial f}{\partial x} = a \in \mathbb{R} \tag{VII.4}$$
- Gradient:
$$f : \mathbb{R}^n \to \mathbb{R}, \qquad x \in \mathbb{R}^n, \quad \frac{\partial f}{\partial x} = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix} = \begin{bmatrix} a_1 \\ \vdots \\ a_n \end{bmatrix} \in \mathbb{R}^n \tag{VII.5}$$
- Gradient:
$$f : \mathbb{R}^{n \times m} \to \mathbb{R}, \qquad x \in \mathbb{R}^{n \times m}, \quad \frac{\partial f}{\partial x} = \begin{bmatrix} \frac{\partial f}{\partial x_{1,1}} & \cdots & \frac{\partial f}{\partial x_{1,m}} \\ \vdots & \ddots & \cdots \\ \frac{\partial f}{\partial x_{n,1}} & \cdots & \frac{\partial f}{\partial x_{n,m}} \end{bmatrix} \in \mathbb{R}^{n \times m} \tag{VII.6}$$
- Jacobian:
$$f : \mathbb{R}^n \to \mathbb{R}^m, \quad f(\begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}) = \begin{bmatrix} f_1 \\ \vdots \\ f_m \end{bmatrix}, \quad \frac{\partial f}{\partial x} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \in \mathbb{R}^{m \times n} \tag{VII.7}$$
- Note that if x was a row vector and so was the function 'image' (result), then this Jacobian matrix would have been transposed.

- An ugly Jacobian (Tensor - A multidimensional matrix):

$$f : \mathbb{R}^{n\times m} \to \mathbb{R}^n, \quad f\left(\begin{bmatrix} w_{11} & \dots & w_{1m} \\ \vdots & \ddots & \vdots \\ w_{n1} & \dots & w_{mn} \end{bmatrix}\right) = \begin{bmatrix} f_1 \\ \vdots \\ f_n \end{bmatrix}, \quad \frac{\partial f}{\partial w} = \begin{bmatrix} \begin{bmatrix} \frac{\partial f_1}{\partial w_{11}} & \dots & \frac{\partial f_1}{\partial w_{1m}} \\ \vdots & \ddots & \dots \\ \frac{\partial f_1}{\partial w_{n1}} & \dots & \frac{\partial f_1}{\partial w_{mn}} \end{bmatrix} \\ \vdots \\ \begin{bmatrix} \frac{\partial f_n}{\partial w_{11}} & \dots & \frac{\partial f_n}{\partial w_{1m}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial w_{n1}} & \dots & \frac{\partial f_n}{\partial w_{mn}} \end{bmatrix} \end{bmatrix} \tag{VII.8}$$

- What is a gradient? It is the derivative scalar-valued differentiable function by a vector or a matrix input.
- **Super important**: Neural networks in general could take any shape of input, but they all result in a loss function, that gives a scalar $L \in \mathbb{R}$. That means:
- In the neural network backpropagation algorithm, we observe only the **current** layer at a time, as an abstraction. We do not try to think of the entire network at once, but step-by-step. Example:

  Toy-Network:

  - Affine()
  - ReLU()
  - **Affine()**
  - Sigmoid()
  - Loss()

  When it comes to think of how to derive the current Affine() layer, we observe it as if it was a function with its own scope.

```
def affine_backward(dout, cache):

    x, w, b = cache
    dx, dw, db = None, None, None

    # some math

    return dx, dw, db
```

According to the chain rule, the derivative of the loss function value L according to the weight matrix of our current affine layer W, would be:

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial \sigma(Y)} \oplus \frac{\partial \sigma(Y)}{\partial Y} \oplus \frac{\partial Y}{\partial W} \tag{VII.9}$$

In this case, $\frac{\partial L}{\partial \sigma(Y)} \frac{\partial \sigma(Y)}{\partial Y}$ is what we call **dout**, or the **upstream gradient**, and we assume it is already calculated before, as in our current scope (according to the relevant functions, of course). Now it is sent to our current scope, to be calculated as a part of the chain-rule, and sent up the stream to the next layer.

Also, $\oplus$ in scalar derivatives represents a simple multiplication. However, in multidimensional derivatives, $\oplus$ represents some unknown function, which we need to figure out.

- In the backpropagation step, the derivative of a learnable weight w_{ij} is to be calculated as a **scalar derivative**:

$$\frac{\partial L}{\partial w_{u,v}} = \sum_i \sum_j \frac{\partial L}{\partial y_{i,j}} \frac{\partial y_{i,j}}{\partial w_{u,v}} \tag{VII.10}$$

Where i, j correspond to the rows and columns of $\frac{\partial L}{\partial Y}$ in some function that utilizes w, such in $Y = f(X) = XW + b$.

Example:

Let $f(X) = 2X$ and $g(X) = \sum_i x_i$. $L = g(f(X))$, where $X \in \mathbb{R}^n$. Therefore:

$$z = f(X)$$

,

$$L = g(f(X)) = g(z) = z_1 + \ldots + z_n = 2x_1 + \ldots + 2x_n$$

$$\begin{aligned}
\frac{\partial L}{\partial x_1} &= \frac{\partial L}{\partial f(x)_1}\frac{\partial f(x)_1}{\partial x_1} + \frac{\partial L}{\partial f(x)_2}\frac{\partial f(x)_2}{\partial x_1} + \ldots + \frac{\partial L}{\partial f(x)_n}\frac{\partial f(x)_n}{\partial x_1} \\
&= 2 \cdot \frac{\partial x1}{\partial x_1} + 2 \cdot \frac{\partial x2}{\partial x_1} + \ldots + 2 \cdot \frac{\partial x_n}{\partial x_1} \\
&= 2 \cdot 1 + 2 \cdot 0 + \ldots + 2 \cdot 0 \\
&= 2
\end{aligned}$$

VII.1.c Stanford Article

- Original article by Stanford (cs231n): Link

Let's follow their example:

$$X = \begin{pmatrix} x_{1,1} & x_{1,2} \\ x_{2,1} & x_{2,2} \end{pmatrix}_{2\times 2} \quad W = \begin{pmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \end{pmatrix}_{2\times 3} \tag{VII.11}$$

$$Y = XW = \begin{pmatrix} x_{1,1}w_{1,1} + x_{1,2}w_{2,1} & x_{1,1}w_{1,2} + x_{1,2}w_{2,2} & x_{1,1}w_{1,3} + x_{1,2}w_{2,3} \\ x_{2,1}w_{1,1} + x_{2,2}w_{2,1} & x_{2,1}w_{1,2} + x_{2,2}w_{2,2} & x_{2,1}w_{1,3} + x_{2,2}w_{2,3} \end{pmatrix} \tag{VII.12}$$

Given a loss function $Loss(Y) = L$, we want to calculate $\frac{\partial L}{\partial X}$ or $\frac{\partial L}{\partial W}$.

As seen in (VII.6), the derivative of a scalar by a matrix, is a gradient / Jacobian matrix that has the same shape as the input. Moreover, we saw in (VII.10), that the final derivative of the loss value L by **any** entry of any matrix in the whole neural network is just a **scalar**. For better understanding we could look at the computational graph of the network in, Figure VII.1, to clearly see that each neuron collects and sums the upstream derivatives (from the loss up to it) - that it took part in calculation of, during the forward pass.

$$\frac{\partial L}{\partial Y} = \begin{pmatrix} \frac{\partial L}{\partial y_{1,1}} & \frac{\partial L}{\partial y_{1,2}} & \frac{\partial L}{\partial y_{1,3}} \\ \frac{\partial L}{\partial y_{2,1}} & \frac{\partial L}{\partial y_{2,2}} & \frac{\partial L}{\partial y_{2,3}} \end{pmatrix} \tag{VII.13}$$

So, from (VII.6) we know that the gradient of Y will have the same shape of Y, because L is a scalar, and it is calculated as a part of the chain-rule. This is the abstraction notion that is discussed above.

Let's derive W. Eventually, after the chain-rule, the derivative of W would have the same shape:

$$\frac{\partial L}{\partial W} = \begin{pmatrix} \frac{\partial L}{\partial w_{1,1}} & \frac{\partial L}{\partial w_{1,2}} & \frac{\partial L}{\partial w_{1,3}} \\ \frac{\partial L}{\partial w_{2,1}} & \frac{\partial L}{\partial w_{2,2}} & \frac{\partial L}{\partial w_{2,3}} \end{pmatrix} \tag{VII.14}$$

Now, this is important. We **do not (!!)** want to calculate the Jacobians. For a better explanation why, refer to the attached article. We have also learned that each entry of $\frac{\partial L}{\partial W}$ is a scalar, that is computed as in (VII.10).

So let's divide and conquer. It is always a better practice, because it's hard to wrap our minds on something bigger than scalars.

$$\frac{\partial L}{\partial w_{11}} = \sum_{i=1}^{2}\sum_{j=1}^{3} \frac{\partial L}{\partial y_{ij}} \frac{\partial y_{ij}}{\partial w_{11}} \tag{VII.15}$$

For better visualization, we could look at it as a **dot product**, which is elementwise multiplication and then summation off all cells (Not what we know as np.dot() - this is confusing). Remember: when deriving a function, it is by the input variable (at least one):

$$\frac{\partial L}{\partial w_{11}} = \frac{\partial L}{\partial Y} \cdot \frac{\partial Y}{\partial w_{11}} = \begin{pmatrix} \frac{\partial L}{\partial y_{1,1}} & \frac{\partial L}{\partial y_{1,2}} & \frac{\partial L}{\partial y_{1,3}} \\ \frac{\partial L}{\partial y_{2,1}} & \frac{\partial L}{\partial y_{2,2}} & \frac{\partial L}{\partial y_{2,3}} \end{pmatrix} \begin{pmatrix} \frac{\partial y_{1,1}}{\partial w_{1,1}} & \frac{\partial y_{1,2}}{\partial w_{1,1}} & \frac{\partial y_{1,3}}{\partial w_{1,1}} \\ \frac{\partial y_{2,1}}{\partial w_{1,1}} & \frac{\partial y_{2,2}}{\partial w_{1,1}} & \frac{\partial y_{2,3}}{\partial w_{1,1}} \end{pmatrix} \tag{VII.16}$$

If we go back to (VII.12), we get:

$$\frac{\partial L}{\partial w_{11}} = \begin{pmatrix} \frac{\partial L}{\partial y_{1,1}} & \frac{\partial L}{\partial y_{1,2}} & \frac{\partial L}{\partial y_{1,3}} \\ \frac{\partial L}{\partial y_{2,1}} & \frac{\partial L}{\partial y_{2,2}} & \frac{\partial L}{\partial y_{2,3}} \end{pmatrix} \cdot \begin{pmatrix} x_{1,1} & 0 & 0 \\ x_{2,1} & 0 & 0 \end{pmatrix} \tag{VII.17}$$

Now let's perform the dot product, and we get:

$$\frac{\partial L}{\partial w_{11}} = \frac{\partial L}{\partial y_{1,1}} x_{1,1} + \frac{\partial L}{\partial y_{2,1}} x_{2,1} \tag{VII.18}$$

We can do that for every entry $w_{i,j}$ in W, and we get:

$$\frac{\partial L}{\partial W} = \begin{pmatrix} \frac{\partial L}{\partial y_{1,1}} x_{1,1} + \frac{\partial L}{\partial y_{2,1}} x_{2,1} & \frac{\partial L}{\partial y_{1,2}} x_{1,1} + \frac{\partial L}{\partial y_{2,2}} x_{2,1} & \frac{\partial L}{\partial y_{1,3}} x_{1,1} + \frac{\partial L}{\partial y_{2,3}} x_{2,1} \\ \frac{\partial L}{\partial y_{1,1}} x_{1,2} + \frac{\partial L}{\partial y_{2,1}} x_{2,2} & \frac{\partial L}{\partial y_{1,2}} x_{1,2} + \frac{\partial L}{\partial y_{2,2}} x_{2,2} & \frac{\partial L}{\partial y_{1,3}} x_{1,2} + \frac{\partial L}{\partial y_{2,3}} x_{2,2} \end{pmatrix} \tag{VII.19}$$

From this matrix, with a little experience, we could derive

$$\frac{\partial L}{\partial W} = \begin{pmatrix} x_{1,1} & x_{2,1} \\ x_{1,2} & x_{2,2} \end{pmatrix} \begin{pmatrix} \frac{\partial L}{\partial y_{1,1}} & \frac{\partial L}{\partial y_{1,2}} & \frac{\partial L}{\partial y_{1,3}} \\ \frac{\partial L}{\partial y_{2,1}} & \frac{\partial L}{\partial y_{2,2}} & \frac{\partial L}{\partial y_{2,3}} \end{pmatrix} = X^T \cdot \frac{\partial L}{\partial Y} \tag{VII.20}$$

We could, of course, do the exact same thing in order to derive X, and we will see that:

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y} \cdot W^T \tag{VII.21}$$

Note: This is only true, because L is a scalar. If we just looked at $Y = XW \rightarrow \frac{\partial Y}{\partial W}$ would be a Jacobian.

What about the bias b term in the affine layer?

We could, or course, do the trick of merging it into X and W, as we saw in the lecture.

If not:

1. $$Y = XW + b$$

   where X_{NxD}, W_{DxM}, b_{1xM} XW_{NxM}

   That means that each b_i in b corresponds to one feature in a row of XW - but to add them like that, it is quite impossible mathematically, right?

   - NumPy uses **broadcasting** to duplicate the row vector b to be a matrix B_{NxM}, by simply copying b N times and stacking them together along the rows, or the 0-axis. But that's just the programming application.
   - Mathematically speaking, it's not $Y = XW + b$, but $Y = XW + 1^N b$, where 1^N is a column vector, such that
     $$\begin{bmatrix} 1_1 \\ \vdots \\ 1_N \end{bmatrix} \begin{bmatrix} b_1 & \dots & b_M \end{bmatrix} = \begin{bmatrix} b_1 & \dots & b_M \\ \vdots & \ddots & \vdots \\ b_1 & \dots & b_M \end{bmatrix}_{N \times M}$$
     That gives us the broadcast that python does by itself, and allows us to actually sum those matrices together.

   Now, one can simply follow the exact same paradigm that we've shown above to solve for b, or we could just look at $1^N b$ as another XW, and do the exact same thing as you did for XW, where 1^N was X and b was W.

2. We see that the derivative of $\frac{\partial L}{\partial b}$ is,
   $$\frac{\partial L}{\partial b} = (1^N)^T \cdot \frac{\partial L}{\partial Y} = \left[\sum_{i=1}^N \frac{\partial L}{\partial y_{i,1}}, \dots, \sum_{i=1}^N \frac{\partial L}{\partial y_{i,M}}\right]$$
   Which in NumPy translates into:
   $$np.sum(dout, axis = 0)$$

VII.1.d Exercise

Given a simple neural network as above:

- **Affine()**
- Sigmoid()
- Loss()

Given again that:

$$X = \begin{pmatrix} x_{1,1} & x_{1,2} \\ x_{2,1} & x_{2,2} \end{pmatrix}_{2\times 2} \quad W = \begin{pmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \end{pmatrix}_{2\times 3} \tag{VII.22}$$

$$Y = XW = \begin{pmatrix} x_{1,1}w_{1,1} + x_{1,2}w_{2,1} & x_{1,1}w_{1,2} + x_{1,2}w_{2,2} & x_{1,1}w_{1,3} + x_{1,2}w_{2,3} \\ x_{2,1}w_{1,1} + x_{2,2}w_{2,1} & x_{2,1}w_{1,2} + x_{2,2}w_{2,2} & x_{2,1}w_{1,3} + x_{2,2}w_{2,3} \end{pmatrix} \tag{VII.23}$$

And $Y = Affine(X)$

1. Show a solution to compute the gradient of the **Sigmoid** layer, w.r.t to the upstream gradient.
2. For a concrete example with numbers, replace sigmoid in the activation function with $f(x) = x^2$ (applied **element-wise**) and use input X and weight matrix W below. So now, our network looks like this:

$$X = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}_{2\times 2} \quad W = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}_{2\times 2}$$

$$\text{Affine} : Y = XW$$

$$\text{Activation} : f(Y) = Y^2$$

$$\text{Loss} : L(f(Y)) = \sum_{i}^{\#Rows} \sum_{j}^{\#Cols} y_{i,j}$$

Find:

(a) The loss value $L(f(Y))$

(b) $\frac{\partial L}{\partial f}$

(c) $\frac{\partial L}{\partial y}$

(d) $\frac{\partial L}{\partial X}$

(e) $\frac{\partial L}{\partial W}$

Hint: Remember, we can use:

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y} \cdot W^T$$

and

$$\frac{\partial L}{\partial W} = X^T \cdot \frac{\partial L}{\partial Y}$$

Solutions:

1. Task: "Gradient of Sigmoid" means the derivative of σ with respect to its input: $\frac{\partial L}{\partial Y} = \frac{\partial L}{\partial \sigma}\frac{\partial \sigma}{\partial Y}$

   $$\sigma(Y) = \begin{pmatrix} \sigma(y_{1,1}) & \sigma(y_{1,2}) & \sigma(y_{1,3}) \\ \sigma(y_{2,1}) & \sigma(y_{2,2}) & \sigma(y_{2,3}) \end{pmatrix}$$

   $$\frac{\partial L}{\partial \sigma} = \begin{pmatrix} \frac{\partial L}{\partial \sigma_{1,1}} & \frac{\partial L}{\partial \sigma_{1,2}} & \frac{\partial L}{\partial \sigma_{1,3}} \\ \frac{\partial L}{\partial \sigma_{2,1}} & \frac{\partial L}{\partial \sigma_{2,2}} & \frac{\partial L}{\partial \sigma_{2,3}} \end{pmatrix}$$

   $$\frac{\partial \sigma}{\partial Y} = \begin{pmatrix} \frac{\partial \sigma_{1,1}}{\partial y_{1,1}} & \frac{\partial \sigma_{1,2}}{\partial y_{1,2}} & \frac{\partial \sigma_{1,3}}{\partial y_{1,3}} \\ \frac{\partial \sigma_{2,1}}{\partial y_{2,1}} & \frac{\partial \sigma_{2,2}}{\partial y_{2,2}} & \frac{\partial \sigma_{2,3}}{\partial y_{2,3}} \end{pmatrix} = \begin{pmatrix} \sigma_{1,1}(1-\sigma_{1,1}) & \sigma_{1,2}(1-\sigma_{1,2}) & \sigma_{1,3}(1-\sigma_{1,3}) \\ \sigma_{2,1}(1-\sigma_{2,1}) & \sigma_{2,2}(1-\sigma_{2,2}) & \sigma_{2,3}(1-\sigma_{2,3}) \end{pmatrix}$$

   For scalars, which all of our loss functions (except for Softmax) are, we can compute this as a dot product i.e. **element-wise multiplication**:

   $$\frac{\partial L}{\partial Y} = \begin{pmatrix} \frac{\partial L}{\partial \sigma_{1,1}} & \frac{\partial L}{\partial \sigma_{1,2}} & \frac{\partial L}{\partial \sigma_{1,3}} \\ \frac{\partial L}{\partial \sigma_{2,1}} & \frac{\partial L}{\partial \sigma_{2,2}} & \frac{\partial L}{\partial \sigma_{2,3}} \end{pmatrix} \cdot \begin{pmatrix} \sigma_{1,1}(1-\sigma_{1,1}) & \sigma_{1,2}(1-\sigma_{1,2}) & \sigma_{1,3}(1-\sigma_{1,3}) \\ \sigma_{2,1}(1-\sigma_{2,1}) & \sigma_{2,2}(1-\sigma_{2,2}) & \sigma_{2,3}(1-\sigma_{2,3}) \end{pmatrix} =$$

   $$= \begin{pmatrix} \frac{\partial L}{\partial \sigma_{1,1}} \cdot \sigma_{1,1}(1-\sigma_{1,1}) & \frac{\partial L}{\partial \sigma_{1,2}} \cdot \sigma_{1,2}(1-\sigma_{1,2}) & \frac{\partial L}{\partial \sigma_{1,3}} \cdot \sigma_{1,3}(1-\sigma_{1,3}) \\ \frac{\partial L}{\partial \sigma_{2,1}} \cdot \sigma_{2,1}(1-\sigma_{2,1}) & \frac{\partial L}{\partial \sigma_{2,2}} \cdot \sigma_{2,2}(1-\sigma_{2,2}) & \frac{\partial L}{\partial \sigma_{2,3}} \cdot \sigma_{2,3}(1-\sigma_{2,3}) \end{pmatrix}$$

   In code we can use the element-wise operator $\oplus$ like this: $\frac{\partial L}{\partial Y} = \frac{\partial L}{\partial \sigma} \oplus \sigma(1-\sigma)$

2. Task:

   (a) Affine: $Y = \begin{pmatrix} x_{1,1}w_{1,1} + x_{1,2}w_{2,1} & x_{1,1}w_{1,2} + x_{1,2}w_{2,2} \\ x_{2,1}w_{1,1} + x_{2,2}w_{2,1} & x_{2,1}w_{1,2} + x_{2,2}w_{2,2} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 5 & 5 \end{pmatrix}$

   Activation: $f(Y) = \begin{pmatrix} y_{1,1}^2 & y_{1,2}^2 \\ y_{2,1}^2 & y_{2,2}^2 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 25 & 25 \end{pmatrix}$

   Loss: $L(f(Y)) = f_{1,1} + f_{1,2} + f_{2,1} + f_{2,2} = 1 + 1 + 25 + 25 = 52$

   (b) $\frac{\partial L}{\partial f} = \begin{pmatrix} \frac{\partial (f_{1,1}+f_{1,2}+f_{2,1}+f_{2,2})}{\partial f_{1,1}} & \frac{\partial (f_{1,1}+f_{1,2}+f_{2,1}+f_{2,2})}{\partial f_{1,2}} \\ \frac{\partial (f_{1,1}+f_{1,2}+f_{2,1}+f_{2,2})}{\partial f_{2,1}} & \frac{\partial (f_{1,1}+f_{1,2}+f_{2,1}+f_{2,2})}{\partial f_{2,2}} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$

   (c) $\frac{\partial L}{\partial Y} = \frac{\partial L}{\partial f}\frac{\partial f}{\partial Y}$

   $$\frac{\partial f}{\partial Y} = \begin{pmatrix} 2y_{1,1} & 2y_{1,2} \\ 2y_{2,1} & 2y_{2,2} \end{pmatrix} = \begin{pmatrix} 2 & 2 \\ 10 & 10 \end{pmatrix}$$

   Remember that since f is an **element-wise** function, the multiplication with the upstream gradient ("dout") is also element-wise.

   $$\frac{\partial L}{\partial Y} = \frac{\partial L}{\partial f}\frac{\partial f}{\partial Y} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 2 & 2 \\ 10 & 10 \end{pmatrix} = \begin{pmatrix} 2 & 2 \\ 10 & 10 \end{pmatrix}$$

   (d) $\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y} \cdot W^T$

$$\begin{pmatrix} 2 & 2 \\ 10 & 10 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 4 & 4 \\ 20 & 20 \end{pmatrix}$$

(e) $\frac{\partial L}{\partial W} = X^T \cdot \frac{\partial L}{\partial Y}$

$$\begin{pmatrix} 0 & 2 \\ 1 & 3 \end{pmatrix} \cdot \begin{pmatrix} 2 & 2 \\ 10 & 10 \end{pmatrix} = \begin{pmatrix} 20 & 20 \\ 32 & 32 \end{pmatrix}$$