

Pytorch

```
# Create a tensor with random values between 0 and 1 with the shape [2, 3, 4]
x = torch.rand(2, 3, 4)
```

#It creates a random tensor with 3x4 matrices across 2 channels.
#The len of the shape object is the dimensionality of the tensor.

Given an input **x**, we define our function by **manipulating** that input, usually by matrix-multiplications with weight matrices and additions with so-called bias vectors. As we manipulate our input, we are automatically creating a **computational graph**. This graph shows how to arrive at our output from our input. PyTorch is a **define-by-run** framework; this means that we can just do our manipulations, and PyTorch will keep track of that graph for us. Thus, we create a dynamic computation graph along the way.

So, to recap: the only thing we have to do is to compute the **output**, and then we can ask PyTorch to automatically get the **gradients**.

Note: Why do we want gradients? Consider that we have defined a function, a neural net, that is supposed to compute a certain output for an input vector **x**. We then define an **error measure** that tells us how wrong our network is; how bad it is in predicting output from input. Based on this error measure, we can use the gradients to **update** the weights **W** that were responsible for the output, so that the next time we present input **x** to our network, the output will be closer to what we want.

The first thing we have to do is to specify which tensors require gradients. By default, when we create a tensor, it does not require gradients. So we set: `x.requires_grad_(True)`
We can perform backpropagation on the computation graph by calling the function `backward()` on the last output, which effectively calculates the gradients for each tensor that has the property `requires_grad=True`. `y.backward()`

Optimization

After defining the model and the dataset, it is time to prepare the optimization of the model. During training, we will perform the following steps:

1. Get a batch from the data loader
2. Obtain the predictions from the model for the batch
3. Calculate the loss based on the difference between predictions and labels
4. Backpropagation: calculate the gradients for every parameter with respect to the loss
5. Update the parameters of the model in the direction of the gradients

For updating the parameters, PyTorch provides the package **torch.optim** that has most popular optimizers implemented. The optimizer provides two useful functions: **optimizer.step()**, and **optimizer.zero_grad()**. The step function updates the parameters based on the gradients as explained above.

The function `optimizer.zero_grad()` sets the gradients of all parameters to zero.

While this function seems less relevant at first, it is a **crucial pre-step** before performing backpropagation. If we call the `backward` function on the loss while the parameter gradients are non-zero from the previous batch, the new gradients would actually be added to the previous ones instead of overwriting them.

This is done because a parameter might occur multiple times in a computation graph, and we need to sum the gradients in this case instead of replacing them. Hence, **remember to call `optimizer.zero_grad()` before calculating the gradients of a batch.**

```
# Set model to train mode
model.train()

# Training loop
for epoch in tqdm(range(num_epochs)):
    for data_inputs, data_labels in data_loader:

        ## Step 1: Move input data to device (only strictly necessary if we use GPU)
        data_inputs = data_inputs.to(device)
        data_labels = data_labels.to(device)

        ## Step 2: Run the model on the input data
        preds = model(data_inputs)
        preds = preds.squeeze(dim=1) # Output is [Batch size, 1], but we want [Batch size]

        ## Step 3: Calculate the loss
        loss = loss_module(preds, data_labels.float())

        ## Step 4: Perform backpropagation
        # Before calculating the gradients, we need to ensure that they are all zero.
        # The gradients would not be overwritten, but actually added to the existing ones.
        optimizer.zero_grad()
        # Perform backpropagation
        loss.backward()

        ## Step 5: Update the parameters
        optimizer.step()
```

When evaluating the model, we don't need to keep track of the computation graph as we don't intend to calculate the gradients. This reduces the required memory and speed up the model. In PyTorch, we can deactivate the computation graph using `with torch.no_grad():`. Remember to additionally set the model to eval mode.

```
model.eval() # Set model to eval mode
true_preds, num_preds = 0., 0.

with torch.no_grad(): # Deactivate gradients for the following code
    for data_inputs, data_labels in data_loader:

        # Determine prediction of model on dev set
        data_inputs, data_labels = data_inputs.to(device), data_labels.to(device)
        preds = model(data_inputs)
        preds = preds.squeeze(dim=1)
        preds = torch.sigmoid(preds) # Sigmoid to map predictions between 0 and 1
        pred_labels = (preds >= 0.5).long() # Binarize predictions to 0 and 1

        # Keep records of predictions for the accuracy metric (true_preds=TP+TN,
        num_preds=TP+TN+FP+FN)
        true_preds += (pred_labels == data_labels).sum()
        num_preds += data_labels.shape[0]

acc = true_preds / num_preds
```

Activation functions

Two of the most popular activation functions are sigmoid and tanh defined as the following:

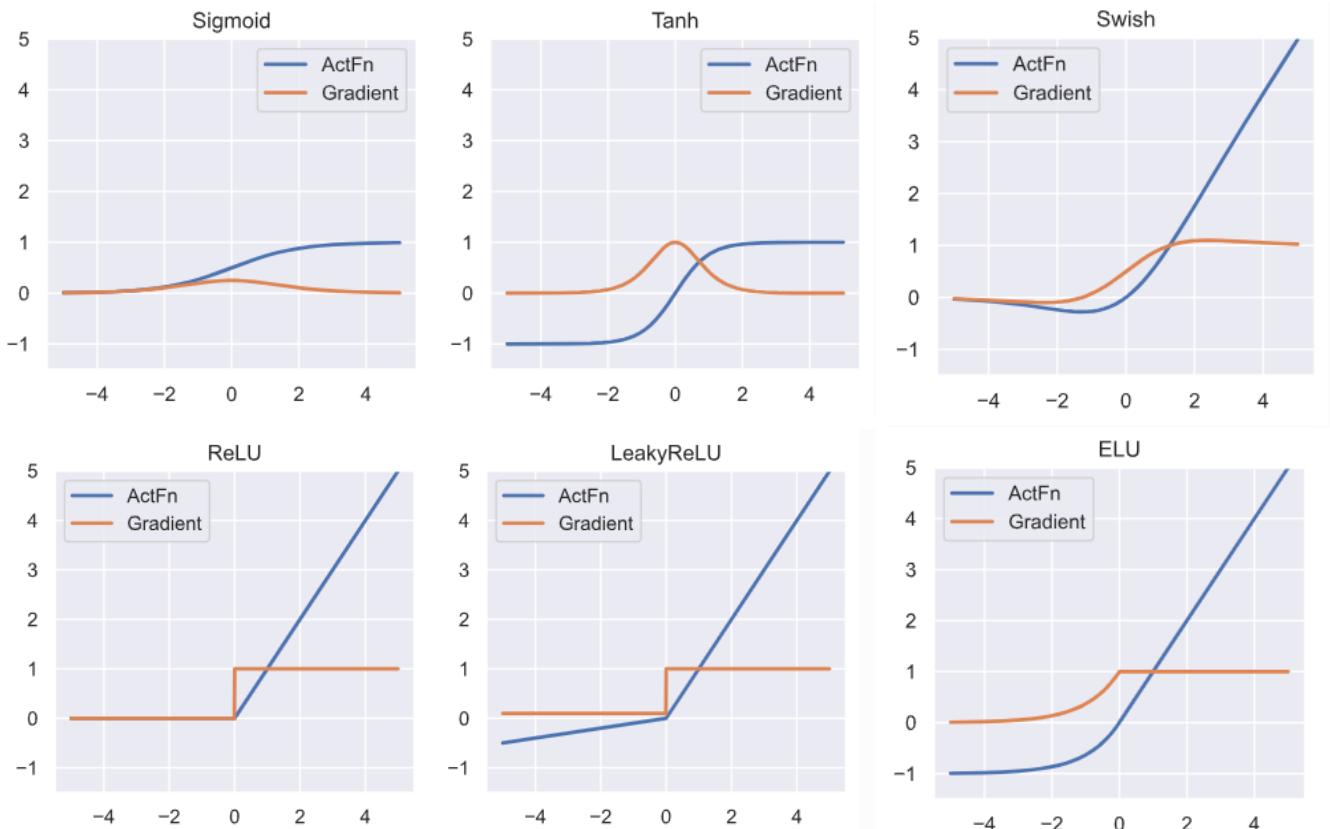
$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}} \quad \tanh = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Another popular activation function that has allowed the training of deeper networks, is the Rectified Linear Unit (ReLU). Despite its simplicity of being a piecewise linear function, ReLU has one major benefit compared to sigmoid and tanh: **a strong, stable gradient for a large range of values**. Based on this idea, a lot of variations of ReLU have been proposed, of which we will implement the following three: **LeakyReLU, ELU, and Swish**. LeakyReLU replaces the zero settings in the negative part with a smaller slope to allow gradients to flow also in this part of the input. Similarly, ELU replaces the negative part with an exponential decay. The third, most recently proposed activation function is Swish, which is actually the result of a large experiment with the purpose of finding the “optimal” activation function. Compared to the other activation functions, Swish is both smooth and non-monotonic (i.e. contains a change of sign in the gradient). This has been shown to prevent dead neurons as in standard ReLU activation, especially for deep networks.

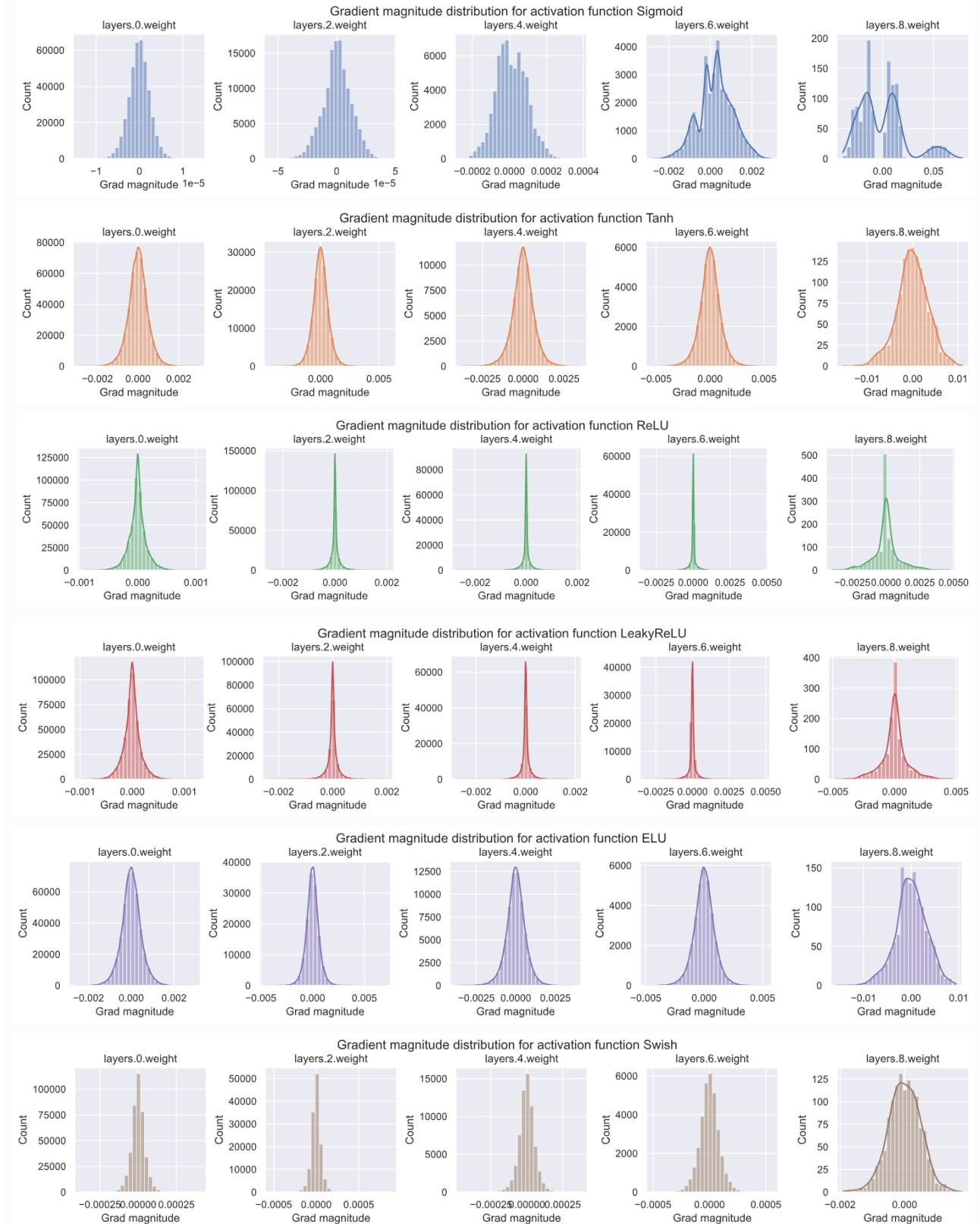
$$\text{ReLU}(x) = \max(0, x)$$

$$\text{LeakyReLU}(x, \alpha) = \begin{cases} x & x \geq 0 \\ \alpha * x & x < 0 \end{cases}$$

$$\text{ELU}(x) = \begin{cases} x & x \geq 0 \\ e^x - 1 & x < 0 \end{cases} \quad \text{Swish}(x) = x * \text{sigmoid}(x)$$



As mentioned previously, one important aspect of activation functions is how they propagate gradients through the network. Imagine we have a very deep neural network with more than 50 layers. The gradients for the input layer, i.e. the very first layer, have passed >50 times the activation function, but we still want them to be of a reasonable size. If the gradient through the activation function is (in expectation) considerably smaller than 1, our gradients will vanish until they reach the input layer. If the gradient through the activation function is larger than 1, the gradients exponentially increase and might explode.



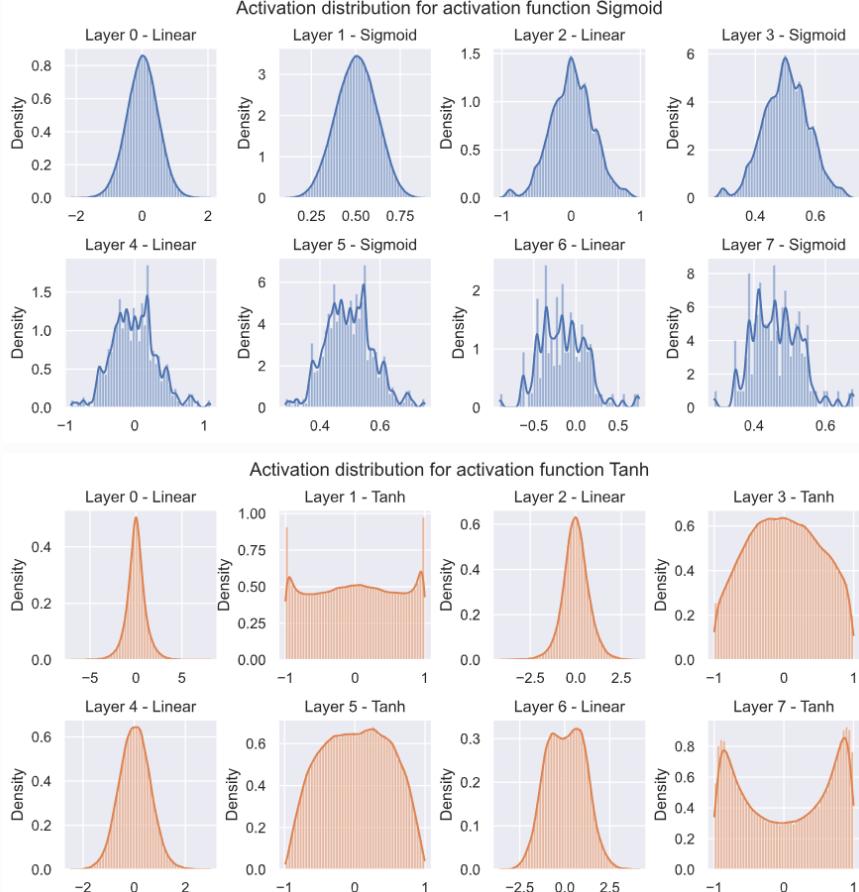
The sigmoid activation function shows a clearly undesirable behavior. While the gradients for the output layer are very large with up to 0.1, the input layer has the lowest gradient norm across all activation functions with only 1e-5. This is due to its small maximum gradient of 1/4, and finding a suitable learning rate across all layers is not possible in this setup. All the other activation functions show to have similar gradient norms across all layers. Interestingly, **the ReLU activation has a spike around 0 which is caused by its zero-part on the left**, and **dead neurons** (we will take a closer look at this later on).

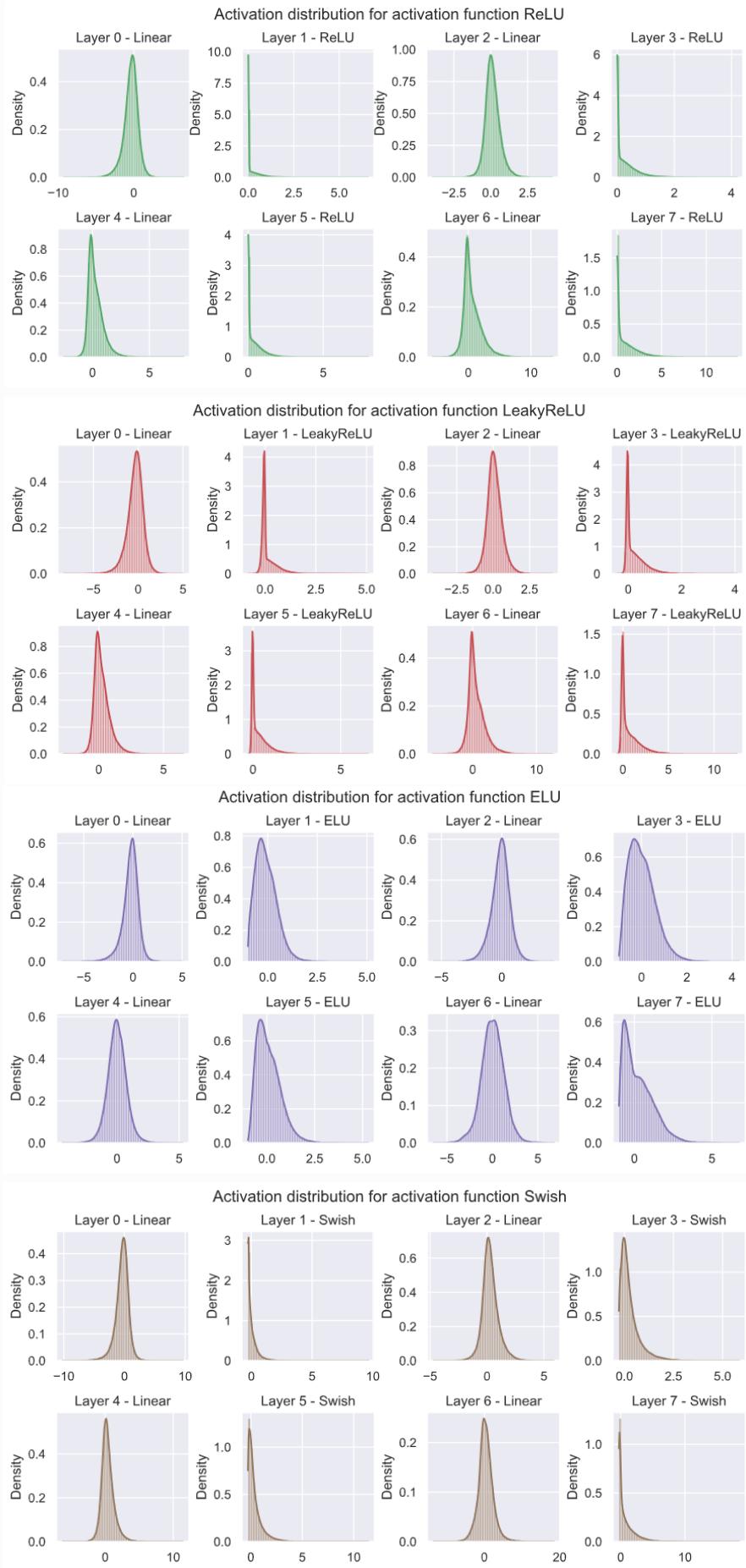
Note that additionally to the activation, the initialization of the weight parameters can be crucial. By default, PyTorch uses the **Kaiming initialization for linear layers optimized for ReLU activations**. In Tutorial 4, we will take a closer look at initialization, but assume for now that the Kaiming initialization works for all activation functions reasonably well.

When training, the model using the sigmoid activation function shows to fail and does not improve upon random performance (10 classes => 1/10 for random chance).

All the other activation functions gain similar performance. To have a more accurate conclusion, we would have to train the models for multiple seeds and look at the averages. However, **the “optimal” activation function also depends on many other factors** (hidden sizes, number of layers, type of layers, task, dataset, optimizer, learning rate, etc.) so that a thorough grid search would not be useful in our case. In the literature, activation functions that have shown to work well with deep networks are all types of ReLU functions we experiment with here, with small gains for specific activation functions in specific networks.

After we have trained the models, we can look at the actual activation values that find inside the model. For instance, how many neurons are set to zero in ReLU? Where do we find most values in Tanh?





As the model with sigmoid activation was not able to train properly, the activations are also less informative and all gathered around 0.5 (the activation at input 0).

The tanh shows a more diverse behavior. While for the input layer we experience a larger amount of neurons to be close to -1 and 1, where the gradients are close to zero, the activations in the two consecutive layers are closer to zero. This is probably because the input layers look for specific features in the input image, and the consecutive layers combine those together. The activations for the last layer are again more biased to the extreme points because the classification layer can be seen as a weighted average of those values (the gradients push the activations to those extremes).

The ReLU has a strong peak at 0, as we initially expected. The effect of having no gradients for negative values is that the network does not have a Gaussian-like distribution after the linear layers, but a longer tail towards the positive values. The LeakyReLU shows a very similar behavior while ELU follows again a more Gaussian-like distribution. The Swish activation seems to lie in between, although it is worth noting that Swish uses significantly higher values than other activation functions (up to 20).

As all activation functions show slightly different behavior although obtaining similar performance for our simple network, it becomes apparent that the selection of the “optimal” activation function really depends on many factors, and is not the same for all possible networks.

Dead neurons

One known drawback of the ReLU activation is the occurrence of “dead neurons”, i.e. neurons with no gradient for any training input. The issue of dead neurons is that as no gradient is provided for the layer, we cannot train the parameters of this neuron in the previous layer to obtain output values besides zero. **For dead neurons to happen, the output value of a specific neuron of the linear layer before the ReLU has to be negative for all input images.** Considering the large number of neurons we have in a neural network, it is not unlikely for this to happen.

The number of dead neurons usually increase with the depth of the layer. However, it should be noted that dead neurons are especially problematic in the input layer. As the input does not change over epochs (the training set is kept as it is), training the network cannot turn those neurons back active. Still, the input data has usually a sufficiently high standard deviation to reduce the risk of dead neurons.

Optimization

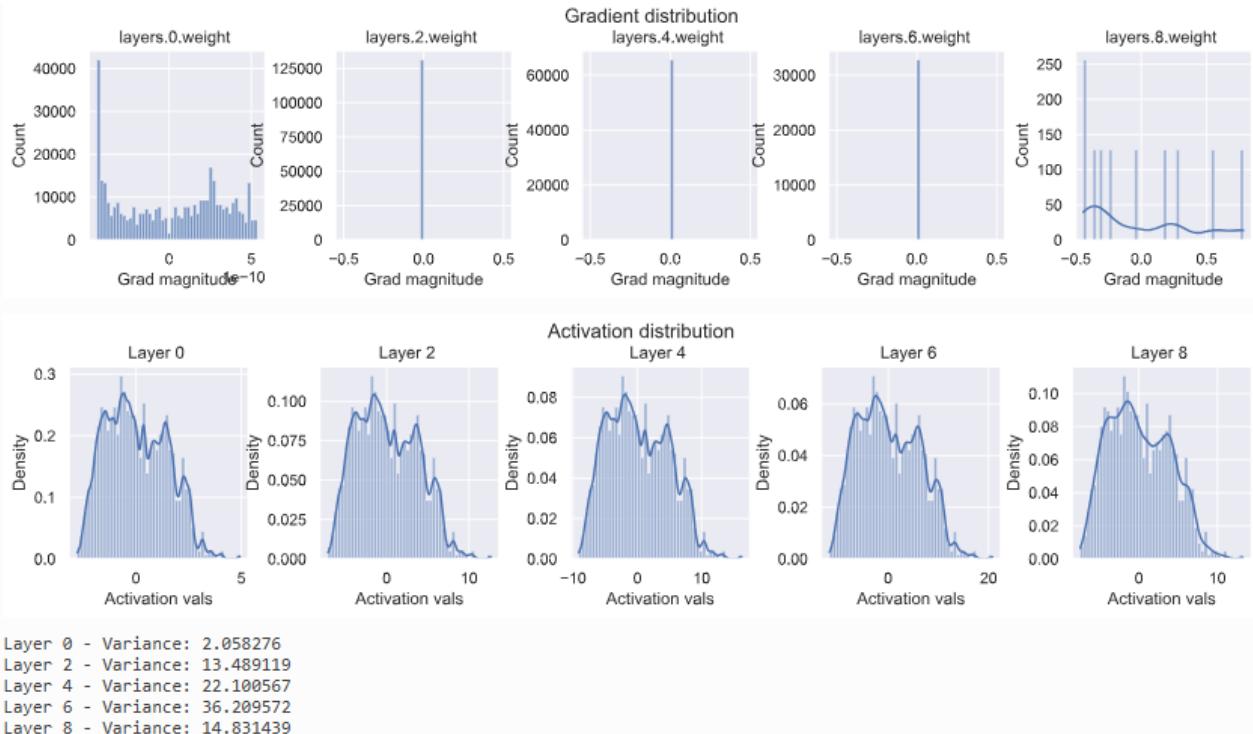
Initialization

When initializing a neural network, there are a few properties we would like to have. First, **the variance of the input should be propagated through the model to the last layer**, so that we have a similar standard deviation for the output neurons. **If the variance would vanish** the deeper we go in our model, **it becomes much harder to optimize** the model as the input to the next layer is basically a single constant value. Similarly, **if the variance increases, it is likely to explode** (i.e. head to infinity) the deeper we design our model. The **second property** we look out for in initialization techniques is a **gradient distribution with equal variance** across layers. **If the first layer receives much smaller gradients than the last layer, we will have difficulties in choosing an appropriate learning rate.**

As a starting point for finding a good method, we will analyze different initialization based on our linear neural network with no activation function (i.e. an identity). We do this because initializations depend on the specific activation function used in the network, and we can adjust the initialization schemes later on for our specific choice.

Constant initialization

The first initialization we can consider is to initialize all weights with the same constant value. Intuitively, setting all weights to zero is not a good idea as the propagated gradient will be zero. However, what happens if we set all weights to a value slightly larger or smaller than 0? To find out, we can implement a function for setting all parameters below and visualize the gradients.



As we can see, only the first and the last layer have diverse gradient distributions while the other three layers have the same gradient for all weights (note that this value is unequal 0, but often very close to it). **Having the same gradient for parameters that have been initialized with the same values means that we will always have the same value for those parameters.** Since every neuron output the same value, they all receive the same error signal from the subsequent layer (assuming the next layer also has symmetric weights or is the

output layer with a symmetric loss structure). Since the gradients are identical, the weight update will be identical for every neuron. This would make our layer useless and reduce our effective number of parameters to 1. Thus, we cannot use a constant initialization to train our networks.

Constant variance

From the experiment above, we have seen that a constant value is not working. So instead, how about we initialize the parameters by randomly sampling from a distribution like a Gaussian? The most intuitive way would be to choose one variance that is used for all layers in the network.

How to find appropriate initialization values

From our experiments above, we have seen that we need to sample the weights from a distribution, but are not sure which one exactly. As a next step, we will try to find the optimal initialization from the perspective of the activation distribution. For this, we state two requirements:

1. The mean of the activations should be zero.
2. The variance of the activations should stay the same across every layer

Our goal is that the variance of each output element of a layer is the same as the input. We assume \mathbf{x} to have mean zero, since the output is the input of another layer this requires the bias and weight to have an expectation of 0. Actually, as \mathbf{b} is a single element per output neuron and is constant across different inputs, we set it to 0 overall.

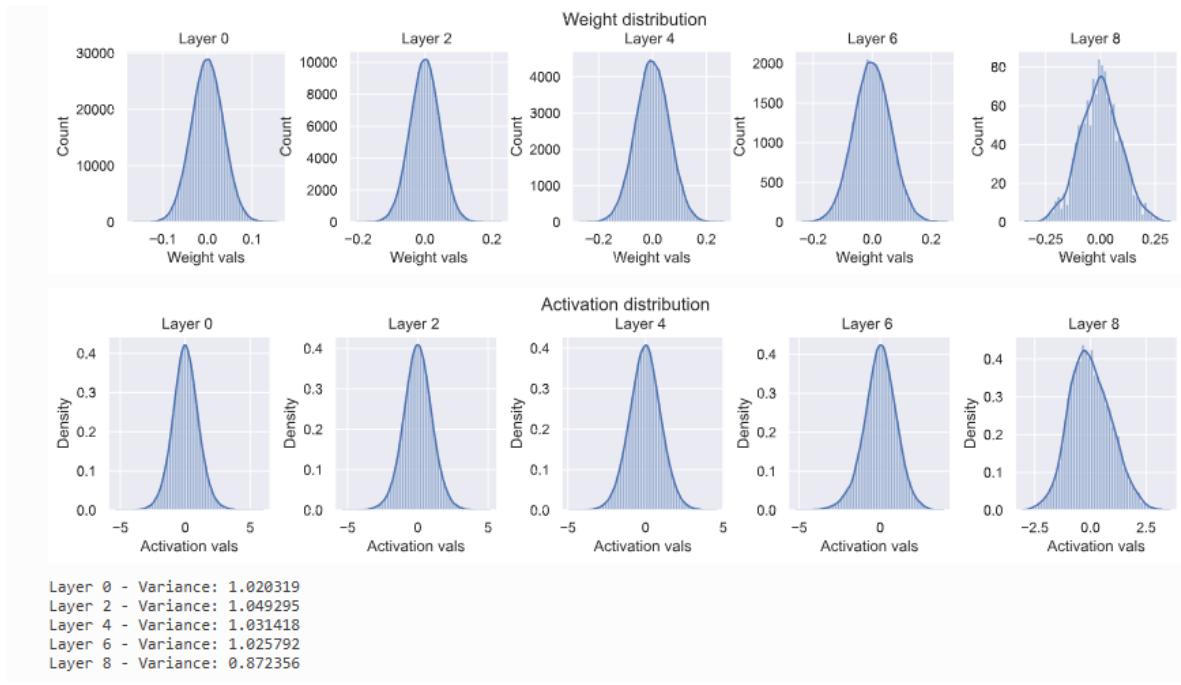
Next, we need to calculate the variance with which we need to initialize the weight parameters. Along the calculation, we will need the following variance rule: given two independent variables, the variance of their product is:

$$\begin{aligned} \text{Var}(X, Y) &= [Y]^2 \text{Var}(X) + EX[X]^2 \text{Var}(Y) + \text{Var}(X) \text{Var}(Y) \\ &= EX[Y^2]EX[X^2] - EX[Y]^2EX[X]^2 \end{aligned}$$

The needed variance of the weights, $\text{Var}(w_{ij})$, is calculated as follows:

$$\begin{aligned} y_i &= \sum_j w_{ij}x_j && \text{Calculation of a single output neuron without bias} \\ \text{Var}(y_i) &= \sigma_x^2 = \text{Var}\left(\sum_j w_{ij}x_j\right) \\ &= \sum_j \text{Var}(w_{ij}x_j) && \text{Inputs and weights are independent of each other} \\ &= \sum_j \text{Var}(w_{ij}) \cdot \text{Var}(x_j) && \text{Variance rule (see above) with expectations being zero} \\ &= d_x \cdot \text{Var}(w_{ij}) \cdot \text{Var}(x_j) && \text{Variance equal for all } d_x \text{ elements} \\ &= \sigma_x^2 \cdot d_x \cdot \text{Var}(w_{ij}) \\ \Rightarrow \text{Var}(w_{ij}) &= \sigma_w^2 = \frac{1}{d_x} \end{aligned}$$

Thus, we should initialize the weight distribution with a variance of the inverse of the input dimension d_x .



As we expected, the variance stays indeed constant across layers. Note that our initialization does not restrict us to a normal distribution, but allows any other distribution with a mean of 0 and variance of $1/d_x$. You often see that a uniform distribution is used for initialization. A small benefit of using a uniform instead of a normal distribution is that we can exclude the chance of initializing very large or small weights.

Nice blog post if you fell confused: [this](#).

Besides the variance of the activations, another **variance we would like to stabilize is the one of the gradients**. This ensures a stable optimization for deep networks. It turns out that we can do the same calculation as above starting from $\Delta x = W \Delta y$, and come to the conclusion that we should initialize our layers with $1/d_y$ where d_y is the number of output neurons. As a compromise between both constraints, [Glorot and Bengio \(2010\)](#) proposed to use the harmonic mean of both values. This leads us to the well-known Xavier initialization:

$$W \sim \mathcal{N} \left(0, \frac{2}{d_x + d_y} \right)$$

If we use a uniform distribution, we would initialize the weights with:

$$W \sim U \left[-\frac{\sqrt{6}}{\sqrt{d_x + d_y}}, \frac{\sqrt{6}}{\sqrt{d_x + d_y}} \right]$$

```
layers.0.weight - Variance: 0.000436
layers.2.weight - Variance: 0.000747
layers.4.weight - Variance: 0.001149
layers.6.weight - Variance: 0.001744
layers.8.weight - Variance: 0.017655
```

```
Layer 0 - Variance: 1.216592
Layer 2 - Variance: 1.719161
Layer 4 - Variance: 1.714506
Layer 6 - Variance: 2.224779
Layer 8 - Variance: 5.297660
```

We see that the Xavier initialization balances the variance of gradients and activations. Note that the significantly higher variance for the output layer is due to the large difference of input and output dimension (128 vs 10). However, we currently assumed the activation function to be linear. So what happens if we add a non-linearity? In a tanh-based network, a common assumption is that for small values during the initial steps in training, the **tanh** works as a linear function such that we don't have to adjust our calculation. We can check if that is the case for us as well:

```
layers.0.weight - Variance: 0.000016
layers.2.weight - Variance: 0.000027
layers.4.weight - Variance: 0.000036
layers.6.weight - Variance: 0.000049
layers.8.weight - Variance: 0.000455
```

```
Layer 0 - Variance: 1.295969
Layer 2 - Variance: 0.583388
Layer 4 - Variance: 0.291432
Layer 6 - Variance: 0.265237
Layer 8 - Variance: 0.274929
```

Although the variance decreases over depth, it is apparent that the activation distribution becomes more focused on the low values. Therefore, our variance will stabilize around 0.25 if we would go even deeper. Hence, we can conclude that the **Xavier initialization works well for Tanh networks**. But what about ReLU networks? Here, **we cannot take the previous assumption of the non-linearity becoming linear for small values**. The ReLU activation function sets (in expectation) half of the inputs to 0 so that also **the expectation of the input is not zero**. However, as long as the expectation of **W** is zero and **b=0**, **the expectation of the output is zero**. The part where the calculation of the ReLU initialization differs from the identity is when determining $\text{Var}(w_{ij}x_j)$

$$\text{Var}(w_{ij}x_j) = \underbrace{\mathbb{E}[w_{ij}^2]}_{=\text{Var}(w_{ij})} \mathbb{E}[x_j^2] - \underbrace{\mathbb{E}[w_{ij}]^2}_{=0} \mathbb{E}[x_j]^2 = \text{Var}(w_{ij})\mathbb{E}[x_j^2]$$

If we assume now that **x** is the output of a ReLU activation (from a previous layer), we can calculate the expectation as follows:

$$\begin{aligned}\mathbb{E}[x^2] &= \mathbb{E}[\max(0, \tilde{y})^2] \\ &= \frac{1}{2}\mathbb{E}[\tilde{y}^2] \quad \tilde{y} \text{ is zero-centered and symmetric} \\ &= \frac{1}{2}\text{Var}(\tilde{y})\end{aligned}$$

Thus, we see that we have an additional factor of 1/2 in the equation, so that our desired weight variance becomes $2/d_x$. This gives us the Kaiming initialization (see [He, K. et al. \(2015\)](#)). Note that the Kaiming initialization does not use the harmonic mean between input and output size. In their paper, they argue that using d_x or d_y both lead to stable gradients throughout the network, and only depend on the overall input and output size of the network. Hence, we can use here only the input d_x .

layers.0.weight - Variance: 0.000075	Layer 0 - Variance: 1.012342
layers.2.weight - Variance: 0.000108	Layer 2 - Variance: 1.092432
layers.4.weight - Variance: 0.000185	Layer 4 - Variance: 1.268176
layers.6.weight - Variance: 0.000444	Layer 6 - Variance: 1.193706
layers.8.weight - Variance: 0.005548	Layer 8 - Variance: 1.760064

The variance stays stable across layers. We can conclude that the Kaiming initialization indeed works well for ReLU-based networks. Note that for Leaky-ReLU etc., we have to slightly adjust the factor of $\sqrt{2}$ in the variance as half of the values are not set to zero anymore.

Optimization

Besides initialization, selecting a suitable optimization algorithm can be an important choice for deep neural networks. First, we need to understand what an optimizer actually does. The optimizer is responsible to update the network's parameters given the gradients. Hence, we effectively implement a function $w^t = f(w^{t-1}, g^t, \dots)$ with w being the parameters and $g^t = \nabla_{w^{(t-1)}} L^{(t)}$ the gradients at time step t . A common, additional parameter to this function is the learning rate, here denoted by η . Usually, the **learning rate can be seen as the “step size”** of the update. A higher learning rate means that we change the weights more in the direction of the gradients, a smaller means we take shorter steps.

As most optimizers only differ in the implementation of f , we can define a template for an optimizer in PyTorch below.

class OptimizerTemplate:

```

def __init__(self, params, lr):
    self.params = list(params)
    self.lr = lr

def zero_grad(self):
    ## Set gradients of all parameters to zero
    for p in self.params:
        if p.grad is not None:
            p.grad.detach_() # For second-order optimizers important
            p.grad.zero_()

@torch.no_grad()
def step(self):
    ## Apply update step to all parameters
    for p in self.params:
        if p.grad is None: # We skip parameters without any gradients
            continue
        self.update_param(p)

def update_param(self, p):
    # To be implemented in optimizer-specific classes
    raise NotImplementedError

```

Stochastic Gradient Descent

$$w^{(t)} = w^{(t-1)} - \eta \cdot g^{(t)}$$

Stochastic Gradient Descent with momentum

$$\begin{aligned} m^{(t)} &= \beta_1 m^{(t-1)} + (1 - \beta_1) \cdot g^{(t)} \\ w^{(t)} &= w^{(t-1)} - \eta \cdot m^{(t)} \end{aligned}$$

Adam (momentum and RMS prop)

$$\begin{aligned} m^{(t)} &= \beta_1 m^{(t-1)} + (1 - \beta_1) \cdot g^{(t)} \\ v^{(t)} &= \beta_2 v^{(t-1)} + (1 - \beta_2) \cdot (g^{(t)})^2 \\ \hat{m}^{(t)} &= \frac{m^{(t)}}{1 - \beta_1^t}, \hat{v}^{(t)} = \frac{v^{(t)}}{1 - \beta_2^t} \\ w^{(t)} &= w^{(t-1)} - \frac{\eta}{\sqrt{\hat{v}^{(t)}} + \epsilon} \circ \hat{m}^{(t)} \end{aligned}$$

Here epsilon is a small constant used to improve numerical stability for very small gradient norms. Remember that **the adaptive learning rate does not replace the learning rate hyperparameter η** , but rather **acts as an extra factor** and ensures that the gradients of various parameters have a similar norm.

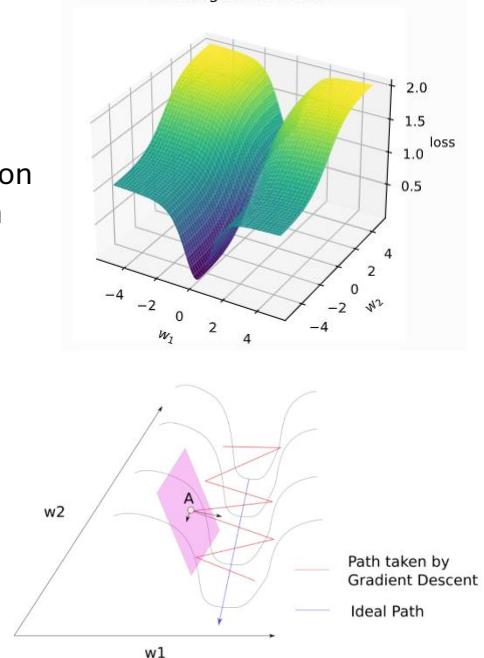
Note that when changing the initialization to worse (e.g. constant initialization), Adam usually shows to be more robust because of its adaptive learning rate. To show the specific benefits of the optimizers, we will continue to look at some possible loss surfaces in which momentum and adaptive learning rate are crucial.

Pathological curvatures

A pathological curvature is a type of surface that is similar to ravines and is particularly tricky for plain SGD optimization. In words, pathological curvatures typically have a steep gradient in one direction with an optimum at the center, while in a second direction we have a slower gradient towards a (global) optimum.

In terms of optimization, you can image that w_1 and w_2 are weight parameters, and the curvature represents the loss surface over the space of w_1 and w_2 . Note that in typical networks, we have many, many more parameters than two, and such curvatures can occur in multi-dimensional spaces as well. Ideally, our optimization algorithm would find the center of the ravine and focuses on optimizing the parameters towards the direction of w_2 . However, if we encounter a point along the ridges, the gradient is much greater in w_1 than w_2 , and we might end up jumping from one side to the other. Due to the large gradients, we would have to reduce our learning rate slowing down learning significantly.

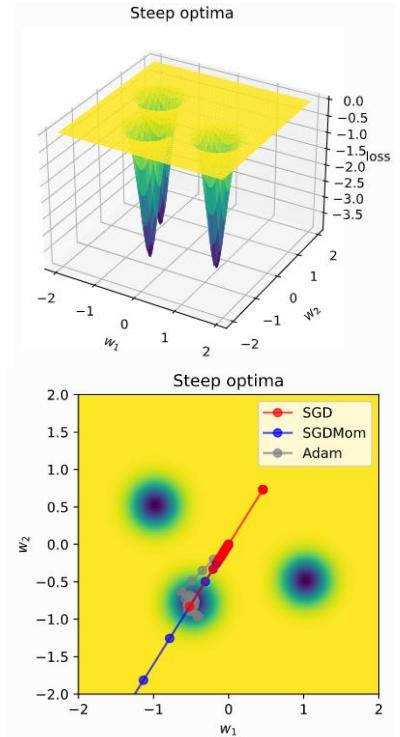
Pathological curvature



Steep optima

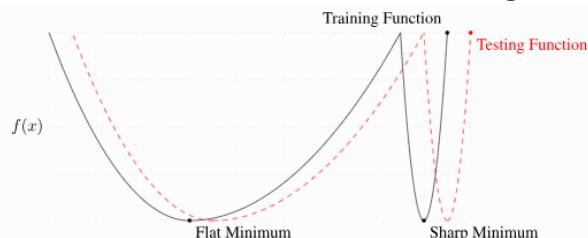
A second type of challenging loss surfaces are steep optima. In those, we have a larger part of the surface having very small gradients while around the optimum, we have very large gradients.

Most of the loss surface has very little to no gradients. However, close to the optima, we have very steep gradients. To reach the minimum when starting in a region with lower gradients, we expect an adaptive learning rate to be crucial. SGD first takes very small steps until it touches the border of the optimum. First reaching a point around $(-0.75, 0.75)$ the gradient direction has changed and pushes the parameters to $(0.8, 0.5)$ from which SGD cannot recover anymore (only with many, many steps). A similar problem has SGD with momentum, only that it continues the direction of the touch of the optimum. The gradients from this time step are so much larger than any other point that the momentum m_t is overpowered by it. Finally, Adam is able to converge in the optimum showing the importance of adaptive learning rates.



After seeing the results on optimization, what is our conclusion?

Should we always use Adam and never look at SGD anymore? The short answer: no. There are many papers saying that in certain situations, SGD (with momentum) generalizes better where Adam often tends to overfit. This is related to the idea of finding wider optima.



The black line represents the training loss surface, while the dotted red line is the test loss.

Finding sharp, narrow minima can be helpful for finding the minimal training loss.

However, this doesn't mean that it also minimizes the test loss as especially flat minima have shown to generalize better. You can imagine that the test dataset has a slightly shifted loss surface due to the different examples than in the training set. A small change can have a significant influence for sharp minima, while flat minima are generally more robust to this change.

In the next tutorial, we will see that some network types can still be better optimized with SGD and learning rate scheduling than Adam. Nevertheless, Adam is the most commonly used optimizer in Deep Learning as it usually performs better than other optimizers, especially for deep networks.

Inception, ResNet and DenseNet

In this tutorial, we will implement and discuss variants of modern CNN architectures. There have been many different architectures been proposed over the past few years. Some of the most impactful ones, and still relevant today, are the following: [GoogleNet](#)/Inception architecture, [ResNet](#), and [DenseNet](#). All of them were state-of-the-art models when being proposed, and the core ideas of these networks are the foundations for most current state-of-the-art architectures. Thus, it is important to understand these architectures in detail and learn how to implement them.

Throughout this tutorial, we will train and evaluate the models on the CIFAR10 dataset. we will use data augmentation during training. This reduces the risk of overfitting and helps CNNs to generalize better. Specifically, we will apply two random augmentations.

First, we will flip each image horizontally by a chance of 50%. The object class usually does not change when flipping an image, and we don't expect any image information to be dependent on the horizontal orientation. This would be however different if we would try to detect digits or letters in an image, as those have a certain orientation.

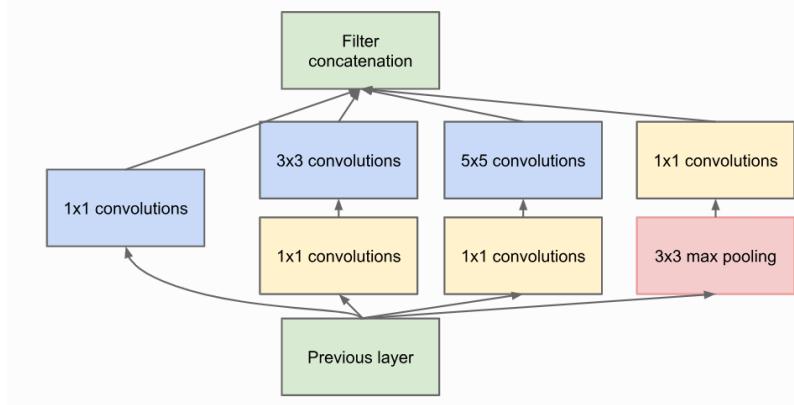
The second augmentation we use is called transforms.RandomResizedCrop. This transformation crops the image in a small range, eventually changing the aspect ratio, and scaling it back afterward to the previous size. Therefore, the actual pixel values change while the content or overall semantics of the image stays the same. We will randomly split the training dataset into a training and a validation set. The validation set will be used for determining early stopping. After finishing the training, we test the models on the CIFAR test set.

Inception

The [GoogleNet](#), proposed in 2014, won the ImageNet Challenge because of its usage of the **Inception modules**. In general, we will mainly focus on the concept of Inception in this tutorial instead of the specifics of the GoogleNet, as based on Inception, there have been many follow-up works ([Inception-v2](#), [Inception-v3](#), [Inception-v4](#), [Inception-ResNet](#),...).

The follow-up works mainly focus on increasing efficiency and enabling very deep Inception networks. However, for a fundamental understanding, it is sufficient to look at the original Inception block.

An Inception block applies four convolution blocks separately on the same feature map: a 1x1, 3x3, and 5x5 convolution, and a max pool operation. **This allows the network to look at the same data with different receptive fields.** Of course, learning only 5x5 convolution would be theoretically more powerful. However, this is not only more computation and memory heavy but also tends to overfit much easier. The overall inception block looks like below:



The additional 1×1 convolutions before the 3×3 and 5×5 convolutions are used for **dimensionality reduction**. This is especially crucial as the feature maps of all branches are merged afterward, and we don't want any explosion of feature size. **As 5×5 convolutions are 25 times more expensive than 1×1 convolutions, we can save a lot of computation and parameters by reducing the dimensionality before the large convolutions.**

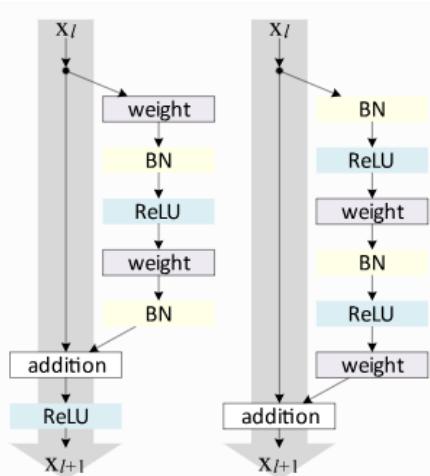
The GoogleNet architecture consists of stacking **multiple Inception** blocks with **occasional max pooling** to reduce the height and width of the feature maps. The original GoogleNet was designed for image sizes of ImageNet (224×224 pixels) and had almost 7 million parameters. As we train on CIFAR10 with image sizes of 32×32 , we don't require such a heavy architecture, and instead, apply a reduced version. The number of channels for dimensionality reduction and output per filter (1×1 , 3×3 , 5×5 , and max pooling) need to be manually specified and can be changed if interested. The general intuition is to have the most filters for the 3×3 convolutions, as they are powerful enough to take the context into account while requiring almost a third of the parameters of the 5×5 convolution.

ResNet

The [ResNet](#) paper is one of the [most cited AI papers](#), and has been the foundation for neural networks with more than 1,000 layers. Despite its simplicity, the idea of residual connections is highly effective as it supports stable gradient propagation through the network. Instead of modeling $x_{l+1} = F(x_l)$ we model $x_{l+1} = x_l + F(x_l)$ where F is a non linear mapping (usually a sequence of NN modules like convolutions, activation functions, and normalizations). If we do backpropagation on such residual connections, we obtain:

$$\frac{\partial x_{l+1}}{\partial x_l} = I + \frac{\partial F(x_l)}{\partial x_l}$$

The bias towards the identity matrix guarantees a stable gradient propagation being less effected by F itself. There have been many variants of ResNet proposed, which mostly concern the function F , or operations applied on the sum. In this tutorial, we look at two of them: the original ResNet block, and the [Pre-Activation ResNet block](#). We visually compare the blocks below

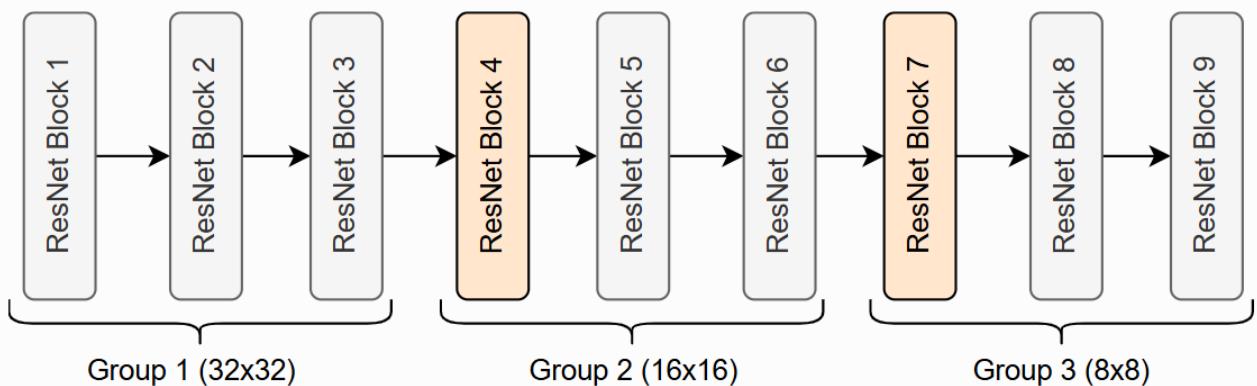


The original ResNet block applies a non-linear activation function, usually ReLU, after the skip connection. In contrast, the pre-activation ResNet block applies the non-linearity at the beginning of F . Both have their advantages and disadvantages. For very deep network, however, **the pre-activation ResNet has shown to perform better** as the gradient flow is guaranteed to have the identity matrix as calculated above, and is not harmed by any non-

linear activation applied to it. For comparison, in this notebook, we implement both ResNet types as shallow networks.

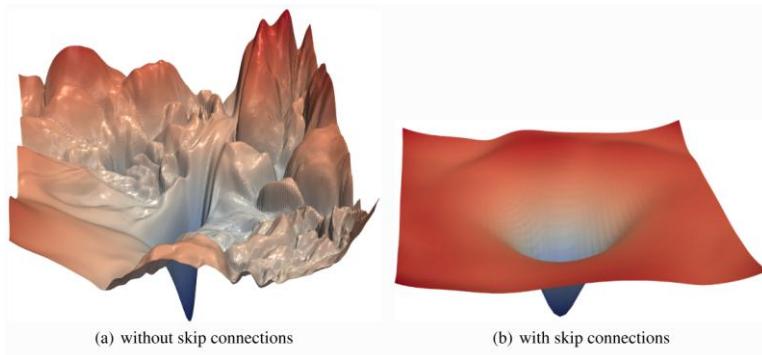
Let's start with the original ResNet block. The visualization above already shows what layers are included in F. One special case we have to handle is when we want to reduce the image dimensions in terms of width and height. The basic ResNet block requires $F(x_l)$ to be of the same shape as x_l . Thus, we need to change the dimensionality of x_l as well before adding it to $F(x_l)$. The original implementation used an identity mapping with stride 2 and padded additional feature dimensions with 0. However, the more common implementation is to use a **1x1 convolution with stride 2** as it allows us to change the feature dimensionality while being efficient in parameter and computation cost.

The overall ResNet architecture consists of stacking multiple ResNet blocks, of which some are downsampling the input. When talking about ResNet blocks in the whole network, we usually group them by the same output shape. Hence, if we say the ResNet has [3,3,3] blocks, it means that we have **3 times a group of 3 ResNet blocks**, where a subsampling is taking place in the fourth and seventh block. The ResNet with [3,3,3] blocks on CIFAR10 is visualized below.



The three groups operate on the resolutions 32x32, 16x16 and 8x8 respectively. The blocks in orange denote ResNet blocks with downsampling.

Finally, we can train our ResNet models. One difference to the GoogleNet training is that we explicitly use **SGD with Momentum as optimizer instead of Adam**. Adam often leads to a slightly worse accuracy on plain, shallow ResNets. It is not 100% clear why Adam performs worse in this context, but one possible explanation is related to ResNet's loss surface. ResNet has been shown to **produce smoother loss surfaces** than networks without skip connection (see [Li et al., 2018](#) for details). A possible visualization of the loss surface with/out skip connections is below:



The x and y axis shows a projection of the parameter space, and the z axis shows the loss values achieved by different parameter values. On smooth surfaces like the one on the right, we might not require an adaptive learning rate as Adam provides. Instead, **Adam can get stuck in local optima while SGD finds the wider minima that tend to generalize better.**

DenseNet

[DenseNet](#) is another architecture for enabling **very deep neural networks** and takes a slightly different perspective on residual connections. **Instead of modeling the difference between layers, DenseNet considers residual connections as a possible way to reuse features across layers**, removing any necessity to learn redundant feature maps. If we go deeper into the network, the model learns abstract features to recognize patterns. However, some complex patterns consist of a combination of abstract features (e.g. hand, face, etc.), and low-level features (e.g. edges, basic color, etc.). **To find these low-level features in the deep layers, standard CNNs have to learn copy such feature maps, which wastes a lot of parameter complexity.** DenseNet provides an efficient way of reusing features by having each convolution depends on all previous input features, but add only a small amount of filters to it. See the figure below for an illustration:

The last layer, called **the transition layer**, is **responsible for reducing the dimensionality of the feature maps** in height, width, and channel size. **Although those technically break the identity backpropagation, there are only a few in a network so that it doesn't affect the gradient flow much.**

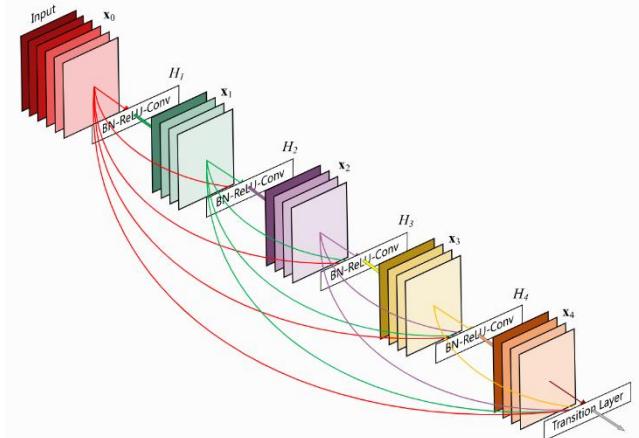
We split the implementation of the layers in DenseNet into three parts: a DenseLayer, and a DenseBlock, and a TransitionLayer.

The module **DenseLayer implements a single layer inside a dense block. It applies a 1x1 convolution for dimensionality reduction with a subsequent 3x3 convolution.** The output channels are concatenated to the originals and returned. Note that **we apply the Batch Normalization as the first layer of each block.** This allows slightly different activations for the same features to different layers, depending on what is needed. Overall, we can implement it as follows:

A dense layer will be:

```
self.net = nn.Sequential(
    nn.BatchNorm2d(c_in),
    act_fn(),
    nn.Conv2d(c_in, bn_size * growth_rate, kernel_size=1, bias=False),
    nn.BatchNorm2d(bn_size * growth_rate),
    act_fn(),
    nn.Conv2d(bn_size * growth_rate, growth_rate, kernel_size=3, padding=1, bias=False))
```

```
def forward(self, x):
    out = self.net(x)
    out = torch.cat([out, x], dim=1)
    return out
```



The module DenseBlock summarizes multiple dense layers applied in sequence. Each dense layer takes as input the original input concatenated with all previous layers' feature maps:

```
layers = []
for layer_idx in range(num_layers):
    layers.append(
        DenseLayer(c_in=c_in + layer_idx * growth_rate, # Input channels are original plus the
feature maps from previous layers
            bn_size=bn_size,
            growth_rate=growth_rate,
            act_fn=act_fn)
    )
self.block = nn.Sequential(*layers)

def forward(self, x):
    out = self.block(x)
    return out
```

Finally, the TransitionLayer takes as input the final output of a dense block and reduces its channel dimensionality using a 1x1 convolution. To reduce the height and width dimension, we take a slightly different approach than in ResNet and apply an average pooling with kernel size 2 and stride 2. This is because we don't have an additional connection to the output that would consider the full 2x2 patch instead of a single value. Besides, it is more parameter efficient than using a 3x3 convolution with stride 2. Thus, the layer is implemented as follows:

```
self.transition = nn.Sequential(
    nn.BatchNorm2d(c_in),
    act_fn(),
    nn.Conv2d(c_in, c_out, kernel_size=1, bias=False),
    nn.AvgPool2d(kernel_size=2, stride=2) # Average the output for each 2x2 pixel group
)
```

We have reviewed four different models. So, which one should we choose if have given a new task? Usually, starting with a ResNet is a good idea given the superior performance of the CIFAR dataset and its simple implementation. Besides, for the parameter number we have chosen here, ResNet is the fastest as DenseNet and GoogleNet have many more layers that are applied in sequence in our primitive implementation. However, if you have a really difficult task, such as semantic segmentation on HD images, more complex variants of ResNet and DenseNet are recommended.

Transformers and Multi-Head Attention

What is Attention?

The attention mechanism describes a recent new group of layers in neural networks that has attracted a lot of interest in the past few years, especially in sequence tasks. There are a lot of different possible definitions of “attention” in the literature, but the one we will use here is the following: *the attention mechanism describes a weighted average of (sequence) elements with the weights dynamically computed based on an input query and elements’ keys.*

So what does this exactly mean? The goal is to take an **average over the features of multiple elements**. However, instead of weighting each element equally, we want to **weight them depending on their actual values**. In other words, we want to **dynamically decide on which inputs we want to “attend” more than others**.

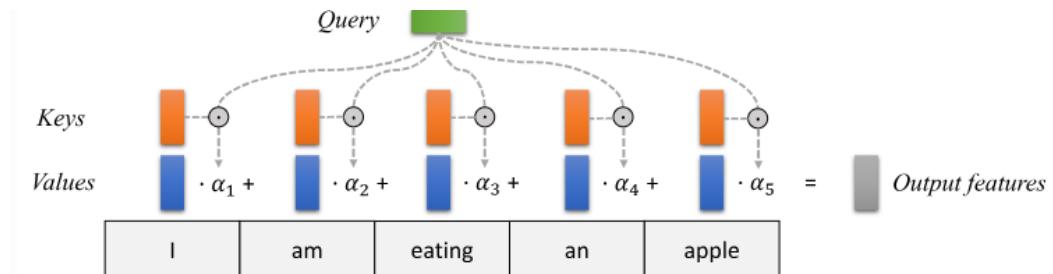
In particular, an attention mechanism has usually four parts we need to specify:

- **Query:** The query is a feature vector that describes what we are looking for in the sequence, i.e. what would we maybe want to pay attention to.
- **Keys:** For each input element, we have a key which is again a feature vector. This feature vector roughly describes what the element is “offering”, or when it might be important. The keys should be designed such that we can identify the elements we want to pay attention to based on the query.
- **Values:** For each input element, we also have a value vector. This feature vector is the one we want to average over.
- **Score function:** To rate which elements we want to pay attention to, we need to specify a score function . The score function takes the query and a key as input, and output the score/attention weight of the query-key pair. **It is usually implemented by simple similarity metrics like a dot product**, or a small MLP.

The weights of the average are calculated by a softmax over all score function outputs. Hence, we assign those value vectors a higher weight whose corresponding key is most similar to the query. If we try to describe it with pseudo-math, we can write:

$$\alpha_i = \frac{\exp(f_{attn}(\text{key}_i, \text{query}))}{\sum_j \exp(f_{attn}(\text{key}_j, \text{query}))}, \quad \text{out} = \sum_i \alpha_i \cdot \text{value}_i$$

Visually, we can show the attention over a sequence of words as follows:



For every word, we have one key and one value vector. The query is compared to all keys with a score function (in this case the dot product) to determine the weights. The softmax is not visualized for simplicity. Finally, the value vectors of all words are averaged using the attention weights.

Most attention mechanisms differ in terms of what queries they use, how the key and value vectors are defined, and what score function is used. The attention applied inside the Transformer architecture is called **self-attention**. In self-attention, **each sequence element** provides a key, value, and query. For each element, we perform an attention layer where based on its query, we check the similarity of the all sequence elements' keys, and returned a different, averaged value vector for each element. We will now go into a bit more detail by first looking at the specific implementation of the attention mechanism which is in the Transformer case **the scaled dot product attention**.

Scaled Dot Product Attention

The core concept behind self-attention is the scaled dot product attention. Our goal is to have an attention mechanism with which any element in a sequence can attend to any other while still being efficient to compute. The dot product attention takes as input a set of queries $Q \in R^{T \times d_k}$, keys $K \in R^{T \times d_k}$, and values $V \in R^{T \times d_v}$ where T is the sequence length, and d_k and d_v are the hidden dimensionality for queries/keys and values respectively. For simplicity, we neglect the batch dimension for now. The attention value from element i to j is based on its similarity of the query Q_i and key K_j , using the dot product as the similarity metric. In math, we calculate the dot product attention as follows:

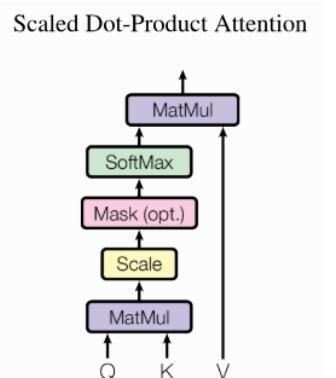
$$\text{Attention}(Q, K, V) = \text{softmax} \left(\frac{QK^T}{\sqrt{d_k}} \right) V$$

The matrix multiplication QK^T performs the dot product for every possible pair of queries and keys, resulting in a matrix of the shape $T \times T$. Each row represents the attention logits for a specific element i to all other elements in the sequence. On these, we apply a softmax and multiply with the value vector to obtain a weighted mean (the weights being determined by the attention). Another perspective on this attention mechanism offers the computation graph which is visualized aside.

One aspect we haven't discussed yet is the scaling factor of $1/\sqrt{d_k}$. This scaling factor is crucial to **maintain an appropriate variance** of attention values after initialization. Remember that we initialize our layers with the intention of having equal variance throughout the model, and hence, Q and K might also have a variance close to 1. However, performing a dot product over two vectors with a variance σ^2 results in a scalar having d_k -times higher variance:

$$q_i \sim \mathcal{N}(0, \sigma^2), k_i \sim \mathcal{N}(0, \sigma^2) \rightarrow \text{Var} \left(\sum_{i=1}^{d_k} q_i \cdot k_i \right) = \sigma^4 \cdot d_k$$

If we do not scale down the variance back to $\sim \sigma^2$, the **softmax over the logits will already saturate** to 1 for one random element and 0 for all others. The gradients through the softmax will be close to zero so that we can't learn the parameters appropriately.



Note that the extra factor of σ^2 , i.e., having σ^4 instead of σ^2 , is usually not an issue, since we keep the original variance σ^2 close to 1 anyways.

The block Mask (opt.) in the diagram above represents the **optional masking** of specific entries in the attention matrix. This is for instance used if we stack multiple sequences with different lengths into a batch. To still benefit from parallelization in PyTorch, **we pad the sentences to the same length and mask out the padding tokens during the calculation of the attention values**. This is usually done by setting the respective attention logits to a very low value.

```
def scaled_dot_product(q, k, v, mask=None):
    d_k = q.size()[-1]
    attn_logits = torch.matmul(q, k.transpose(-2, -1))
    attn_logits = attn_logits / math.sqrt(d_k)
    if mask is not None:
        attn_logits = attn_logits.masked_fill(mask == 0, -9e15)
    attention = F.softmax(attn_logits, dim=-1)
    values = torch.matmul(attention, v)
    return values, attention
```

Multi-Head Attention

The scaled dot product attention allows a network to attend over a sequence. However, often there are **multiple different aspects a sequence element wants to attend to**, and a single weighted average is not a good option for it. This is why we extend the attention mechanisms to **multiple heads**, i.e. **multiple different query-key-value triplets on the same features**. Specifically, given a query, key, and value matrix, we transform those into h sub-queries, sub-keys, and sub-values, which we pass through the scaled dot product attention independently. Afterward, **we concatenate the heads** and combine them with a final weight matrix.

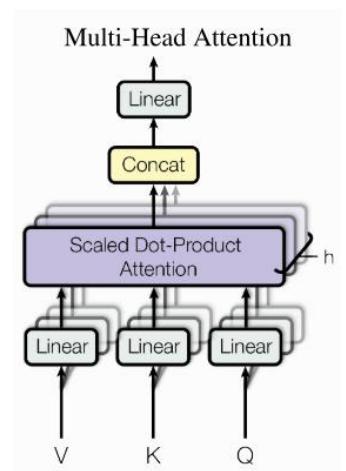
Mathematically, we can express this operation as:

$$\text{Multihead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O$$

$$\text{where } \text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

We refer to this as Multi-Head Attention layer with the learnable parameters $W_{1\dots h}^Q \in \mathbb{R}^{D \times d_k}$, $W_{1\dots h}^K \in \mathbb{R}^{D \times d_k}$, $W_{1\dots h}^V \in \mathbb{R}^{D \times d_v}$, and $W^O \in \mathbb{R}^{h \cdot d_v \times d_{out}}$ (D being the input dimensionality). Expressed in a computational graph, we can visualize it as below (figure credit - [Vaswani et al., 2017](#)).

How are we applying a Multi-Head Attention layer in a neural network, where we don't have an arbitrary query, key, and value vector as input? Looking at the computation graph aside, a simple but effective implementation is to set the current feature map in a NN, $X \in \mathbb{R}^{B \times T \times d_{model}}$ and Q, K, and V (B being the batch size, T the sequence length and d_{model} the hidden dimensionality of X). The consecutive weight matrices W^Q , W^K , and W^V can transform X to the corresponding feature vectors that represent the queries, keys and values of the input.



One crucial characteristic of the multi-head attention is that it is **permutation-equivariant** with respect to its inputs. This means that **if we switch two input elements in the sequence**, e.g. (neglecting the batch dimension for now), **the output is exactly the same** besides the elements 1 and 2 switched. **Hence, the multi-head attention is actually looking at the input not as a sequence, but as a set of elements.** This property makes the multi-head attention block and the Transformer architecture so powerful and widely applicable!

But what if the order of the input is actually important for solving the task, like language modeling? The answer is to **encode the position in the input features**, which we will take a closer look at later (topic *Positional encodings* below).

Before moving on to creating the Transformer architecture, we can compare the self-attention operation with our other common layer competitors for sequence data: convolutions and recurrent neural networks. Below you can find a table by [Vaswani et al. \(2017\)](#) on the complexity per layer, the number of sequential operations, and maximum path length. The complexity is measured by the upper bound of the number of operations to perform, while the maximum path length represents the maximum number of steps a forward or backward signal has to traverse to reach any other position. **The lower this length, the better gradient signals can backpropagate for long-range dependencies.** Let's take a look at the table below:

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(\log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

n is the sequence length, d is the representation dimension and k is the kernel size for convolutions. In contrast to recurrent networks, **the self-attention layer can parallelize all its operations making it much faster to execute for smaller sequence lengths.** However, **when the sequence length exceeds the hidden dimensionality, self-attention becomes more expensive than RNNs.** One way of reducing the computational cost for long sequences is by restricting the self-attention to a neighborhood of inputs to attend over, denoted by r . Nevertheless, there has been recently a lot of work on more efficient Transformer architectures that still allow long dependencies.

Transformer Encoder

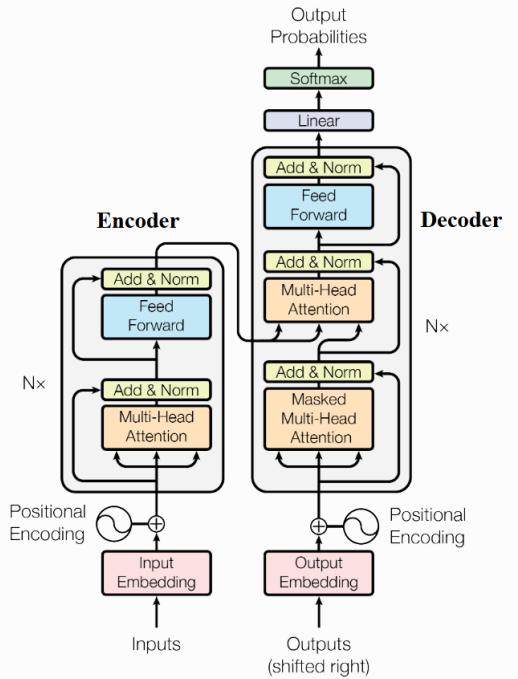
Next, we will look at how to apply the multi-head attention block inside the Transformer architecture. Originally, the Transformer model was designed for machine translation. Hence, it got an **encoder-decoder structure** where the encoder takes as input the sentence in the original language and **generates an attention-based representation**. On the other hand, **the decoder attends over the encoded information and generates the translated sentence in an autoregressive manner**, as in a standard RNN. While this structure is extremely useful for Sequence-to-Sequence tasks with the necessity of autoregressive decoding, we will focus here on the encoder part. Many advances in NLP have been made using pure encoder-based Transformer models (if interested, models include the [BERT](#)-family, the [Vision Transformer](#), and more), and in our tutorial, we will also mainly focus on the encoder part. If you have understood the encoder architecture, the decoder is a very small step to implement as well.

The full Transformer architecture looks as follows:

The encoder consists of N **identical blocks** that are applied in sequence. Taking as input x , it is first passed through a Multi-Head Attention block as we have implemented above. The output is added to the original input using a residual connection, and we apply a consecutive Layer Normalization on the sum.

Overall, it calculates $\text{LayerNorm}(x + \text{MultiHead}(Q, K, V))$. The residual connection is crucial in the Transformer architecture for two reasons:

1. Similar to ResNets, **Transformers are designed to be very deep**. Some models contain more than 24 blocks in the encoder. Hence, **the residual connections are crucial for enabling a smooth gradient flow** through the model.
2. **Without the residual connection, the information about the original sequence is lost**. Remember that the Multi-Head Attention layer ignores the position of elements in a sequence, and can only learn it based on the input features. Removing the residual connections would mean that this information is lost after the first attention layer (after initialization), and with a randomly initialized query and key vector, the output vectors for position has no relation to its original input. All outputs of the attention are likely to represent similar/same information, and there is no chance for the model to distinguish which information came from which input element. **An alternative option to residual connection would be to fix at least one head to focus on its original input**, but this is very inefficient and does not have the benefit of the improved gradient flow.



The **Layer Normalization** also plays an important role in the Transformer architecture as **it enables faster training and provides small regularization**. Additionally, it ensures that the features are in a similar magnitude among the elements in the sequence. We are not using Batch Normalization because it depends on the batch size which is often small with Transformers (they require a lot of GPU memory), and **BatchNorm has shown to perform particularly bad in language as the features of words tend to have a much higher variance** (there are many, very rare words which need to be considered for a good distribution estimate).

Additionally to the Multi-Head Attention, a small fully connected feed-forward network is added to the model, which is applied to each position separately and identically. Specifically, the model uses a **Linear → DropOut → ReLU → Linear** MLP. The full transformation including the residual connection can be expressed as:

$$\begin{aligned} \text{FFN}(x) &= \max(0, xW_1 + b_1)W_2 + b_2 \\ x &= \text{LayerNorm}(x + \text{FFN}(x)) \end{aligned}$$

This MLP adds extra complexity to the model and allows transformations on each sequence element separately. You can imagine as this allows the model to “post-process” the new information added by the previous Multi-Head Attention, and prepare it for the next attention block. Usually, the inner dimensionality of the MLP is 2-8x larger than d_{model} i.e. the dimensionality of the original input x . The general advantage of a wider layer instead of a narrow, multi-layer MLP is the faster, parallelizable execution.

Positional encoding

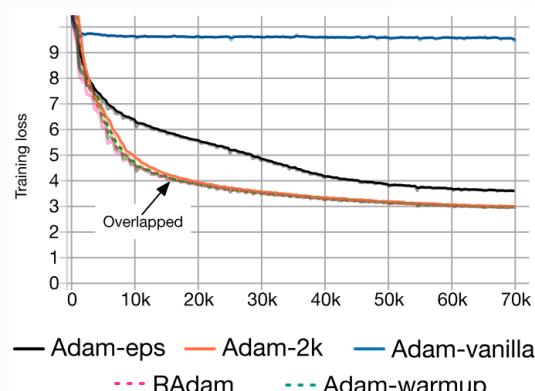
We have discussed before that the **Multi-Head Attention block is permutation-equivariant**, and cannot distinguish whether an input comes before another one in the sequence or not. In tasks like language understanding, however, **the position is important** for interpreting the input words. **The position information can therefore be added via the input features**. We could learn a embedding for every possible position, but this would not generalize to a dynamical input sequence length. Hence, the better option is to use **feature patterns that the network can identify from the features and potentially generalize to larger sequences**. The specific pattern chosen by Vaswani et al. are sine and cosine functions of different frequencies, as follows:

$$PE_{(pos,i)} = \begin{cases} \sin\left(\frac{pos}{10000^{i/d_{model}}}\right) & \text{if } i \bmod 2 = 0 \\ \cos\left(\frac{pos}{10000^{(i-1)/d_{model}}}\right) & \text{otherwise} \end{cases}$$

$PE_{(pos,i)}$ represents the position encoding at position pos in the sequence, and hidden dimensionality i . These values, concatenated for all hidden dimensions, are added to the original input features (in the Transformer visualization above, see “Positional encoding”), and constitute the position information. We distinguish between **even** ($i \bmod 2 = 0$) and **uneven** ($i \bmod 2 = 1$) hidden dimensionalities where we apply a sine/cosine respectively. The intuition behind this encoding is that you can represent $PE_{(pos+k,:)}$ as a linear function of $PE_{(pos,:)}$ which might allow the model to easily attend to relative positions. The wavelength in different dimensions range from 2π to $10000 * 2\pi$.

Learning rate warm-up

One commonly used technique for training a Transformer is learning rate warm-up. This means that we gradually increase the learning rate from 0 on to our originally specified learning rate in the first few iterations. Thus, we slowly start learning instead of taking very large steps from the beginning. In fact, training a deep Transformer without learning rate warm-up can make the model diverge and achieve a much worse performance on training and testing. Take for instance the following plot by [Liu et al. \(2019\)](#) comparing Adam-vanilla.



Clearly, the warm-up is a crucial hyperparameter in the Transformer architecture. Why is it so important? There are currently two common explanations.

Firstly, Adam uses the **bias correction factors which however can lead to a higher variance** in the adaptive learning rate during the first iterations. Improved optimizers like [RAdam](#) have been shown to overcome this issue, not requiring warm-up for training Transformers.

Secondly, **the iteratively applied Layer Normalization across layers can lead to very high gradients during the first iterations**, which can be solved by using [Pre-Layer Normalization](#) (similar to Pre-Activation ResNet), or replacing Layer Normalization by other techniques ([Adaptive Normalization](#), [Power Normalization](#)).

Nevertheless, many applications and papers still use the original Transformer architecture with Adam, because warm-up is a simple, yet effective way of solving the gradient problem in the first iterations. **There are many different schedulers we could use. For instance, the original Transformer paper used an exponential decay scheduler with a warm-up.** However, the currently most popular scheduler is the cosine warm-up scheduler, which combines warm-up with a cosine-shaped learning rate decay. Where in the first 100 iterations, it increase the learning rate factor from 0 to 1, whereas for all later iterations, it decay it using the cosine wave.

An additional parameter we pass to the trainer is `gradient_clip_val`. This clips the norm of the gradients for all parameters before taking an optimizer step and prevents the model from diverging if we obtain very high gradients at, for instance, sharp loss. For Transformers, gradient clipping can help to further stabilize the training during the first few iterations, and also afterward. The clip value is usually between 0.5 and 10, depending on how harsh you want to clip large gradients.

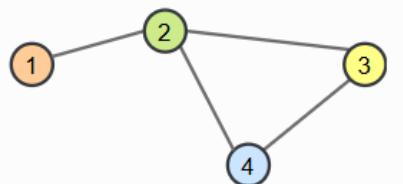
Graph Neural Networks

Graph representation

Before starting the discussion of specific neural network operations on graphs, we should consider how to represent a graph. Mathematically, a graph G is defined as a tuple of a set of nodes/vertices V and a set of edges/links E : $G = (V, E)$. Each edge is a pair of two vertices, and represents a connection between them. For instance, let's look at the following graph:

The vertices are $V = \{1, 2, 3, 4\}$ and edges $E = \{(1, 2), (2, 3), (2, 4), (3, 4)\}$

Note that for simplicity, we assume the graph to be undirected and hence don't add mirrored pairs like $(2, 1)$. In application, vertices and edge can often have specific attributes, and edges can even be directed. The question is how we could represent this diversity in an efficient way for matrix operations. Usually, for the edges, we decide between two variants: an adjacency matrix, or a list of paired vertex indices.



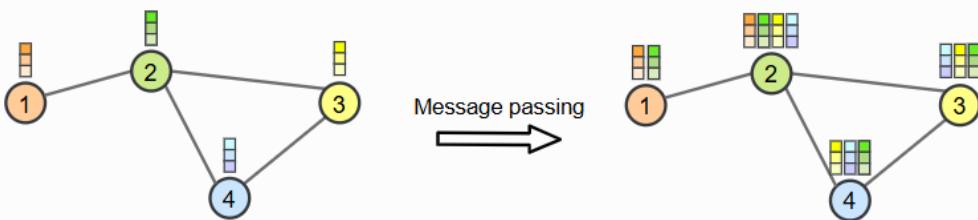
The **adjacency matrix** is a square matrix whose elements indicate whether pairs of vertices are adjacent, i.e. connected, or not. In the simplest case, A_{ij} is 1 if there is a connection from node i to j and otherwise 0. If we have edge attributes or different categories of edges in a graph, this information can be added to the matrix as well. For an undirected graph, keep in mind that **A is a symmetric matrix** ($A_{ij} = A_{ji}$). For the example graph above, we have the following adjacency matrix:

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

While expressing a graph as a list of edges is more efficient in terms of memory and (possibly) computation, using an adjacency matrix is more intuitive and simpler to implement. In our implementations below, we will rely on the adjacency matrix to keep the code simple. However, common libraries use edge lists, which we will discuss later more. Alternatively, we could also use the list of edges to define a sparse adjacency matrix with which we can work as if it was a dense matrix, but allows more memory-efficient operations.

Graph Convolutions

Graph Convolutional Networks have been introduced by [Kipf et al.](#) in 2016 at the University of Amsterdam. He also wrote a great [blog post](#) about this topic, which is recommended if you want to read about GCNs from a different perspective. GCNs are similar to convolutions in images in the sense that the “filter” parameters are typically shared over all locations in the graph. At the same time, GCNs rely on message passing methods, which means that vertices exchange information with the neighbors, and send “messages” to each other. Before looking at the math, we can try to visually understand how GCNs work. The first step is that each node creates a feature vector that represents the message it wants to send to all its neighbors. In the second step, the messages are sent to the neighbors, so that a node receives one message per adjacent node. Below we have visualized the two steps for our example graph.



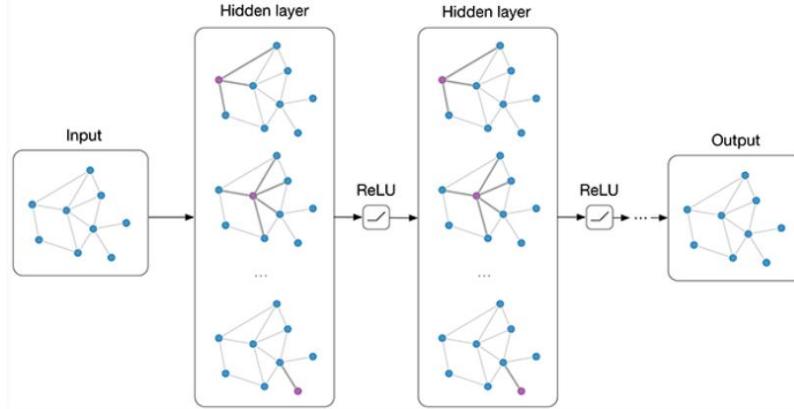
If we want to formulate that in more mathematical terms, we need to first decide how to combine all the messages a node receives. As the number of messages vary across nodes, we need an operation that works for any number. Hence, the usual way to go is to sum or take the mean. Given the previous features of nodes $H^{(l)}$ the GCN layer is defined as follows:

$$H^{(l+1)} = \sigma \left(\hat{D}^{-1/2} \hat{A} \hat{D}^{-1/2} H^{(l)} W^{(l)} \right)$$

$W^{(l)}$ is the weight parameters with which we transform the input features into messages ($H^{(l)} W^{(l)}$). To the adjacency matrix A we add the identity matrix so that each node sends its own message to itself: $\hat{A} = A + I$.

Finally, to take the average instead of summing, we calculate the matrix \widehat{D} which is a diagonal matrix with D_{ii} denoting the number of neighbors node i has. σ represents an arbitrary activation function, and not necessarily the sigmoid (usually a ReLU-based activation function is used in GNNs).

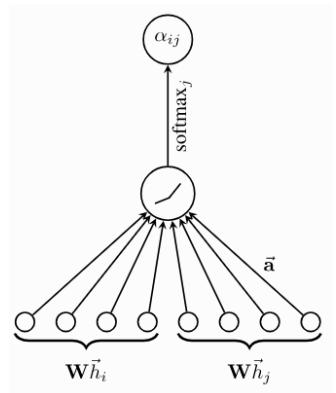
However, in a GNN, we would also want to allow feature exchange between nodes beyond its neighbors. This can be achieved by applying multiple GCN layers, which gives us the final layout of a GNN. The GNN can be built up by a sequence of GCN layers and non-linearities such as ReLU.



However, **one issue that could happen is that the output features for 2 nodes are the same because they have the same adjacent nodes (including itself)**. Therefore, GCN layers can make the network forget node-specific information if we just take a mean over all messages. Multiple possible improvements have been proposed. While **the simplest option might be using residual connections**, the more common approach is to either weigh the self-connections higher or define a separate weight matrix for the self-connections. Alternatively, we can re-visit a concept from the last tutorial: attention.

Graph Attention

If you remember from the last tutorial, attention describes a weighted average of multiple elements with the weights dynamically computed based on an input query and elements' keys. This concept can be similarly applied to graphs, one of such is the Graph Attention Network. **Similarly to the GCN, the graph attention layer creates a message for each node using a linear layer/weight matrix.** For the attention part, it uses the message from the node itself as a query, and the messages to average as both keys and values (note that this also includes the message to itself). The score function is implemented as a one-layer MLP which maps the query and key to a single value. The MLP looks as follows:



h_i and h_j are the original features from node i and j respectively, and represent the messages of the layer with \mathbf{W} as weight matrix. \mathbf{A} is the weight matrix of the MLP, which has the shape $[1, 2 \times d_{message}]$, and α_{ij} the final attention weight from node i to j . The calculation can be described as follows:

$$\alpha_{ij} = \frac{\exp(\text{LeakyReLU}(\mathbf{a}[\mathbf{W}h_i || \mathbf{W}h_j]))}{\sum_{k \in \mathcal{N}_i} \exp(\text{LeakyReLU}(\mathbf{a}[\mathbf{W}h_i || \mathbf{W}h_k]))}$$

The operator $||$ represents the concatenation, and \mathcal{N}_i the indices of the neighbors of node i .

Note that in contrast to usual practice, we apply a non-linearity (here LeakyReLU) before the softmax over elements. Although it seems like a minor change at first, it is crucial for the attention to depend on the original input. Specifically, let's remove the non-linearity for a second, and try to simplify the expression:

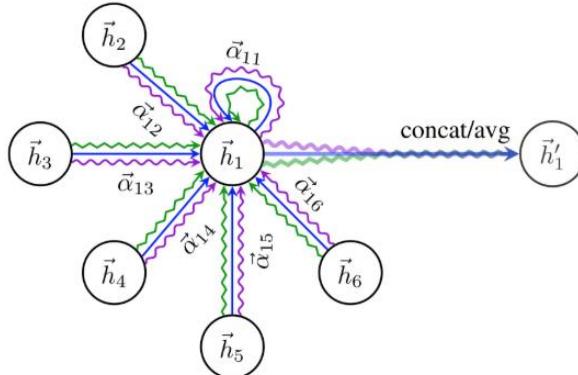
$$\begin{aligned}
 \alpha_{ij} &= \frac{\exp(\mathbf{a}[\mathbf{W}h_i || \mathbf{W}h_j])}{\sum_{k \in \mathcal{N}_i} \exp(\mathbf{a}[\mathbf{W}h_i || \mathbf{W}h_k])} \\
 &= \frac{\exp(\mathbf{a}_{:,d/2} \mathbf{W}h_i + \mathbf{a}_{:,d/2} \mathbf{W}h_j)}{\sum_{k \in \mathcal{N}_i} \exp(\mathbf{a}_{:,d/2} \mathbf{W}h_i + \mathbf{a}_{:,d/2} \mathbf{W}h_k)} \\
 &= \frac{\exp(\mathbf{a}_{:,d/2} \mathbf{W}h_i) \cdot \exp(\mathbf{a}_{:,d/2} \mathbf{W}h_j)}{\sum_{k \in \mathcal{N}_i} \exp(\mathbf{a}_{:,d/2} \mathbf{W}h_i) \cdot \exp(\mathbf{a}_{:,d/2} \mathbf{W}h_k)} \\
 &= \frac{\exp(\mathbf{a}_{:,d/2} \mathbf{W}h_j)}{\sum_{k \in \mathcal{N}_i} \exp(\mathbf{a}_{:,d/2} \mathbf{W}h_k)}
 \end{aligned}$$

We can see that without the non-linearity, the attention term with h_i actually cancels itself out, resulting in the attention being independent of the node itself. Hence, we would have the same issue as the GCN of creating the same output features for nodes with the same neighbors. This is why the LeakyReLU is crucial and adds some dependency on h_i to the attention.

Once we obtain all attention factors, we can calculate the output features for each node by performing the weighted average:

$$h'_i = \sigma \left(\sum_{j \in \mathcal{N}_i} \alpha_{ij} \mathbf{W}h_j \right)$$

σ is yet another non-linearity, as in the GCN layer. Visually, we can represent the full message passing in an attention layer as follows:



To increase the expressiveness of the graph attention network, [Velickovic et al.](#) proposed to **extend it to multiple heads** similar to the Multi-Head Attention block in Transformers. This results in **N attention layers being applied in parallel**. In the image above, it is visualized as three different colors of arrows (green, blue, and purple) that are afterward concatenated. The average is only applied for the very final prediction layer in a network.

Tasks on graph-structured data can be grouped into three groups: node-level, edge-level and graph-level. The different levels describe on which level we want to perform classification / regression.

Node-level tasks: Semi-supervised node classification

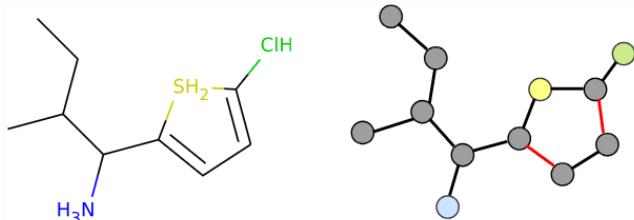
Node-level tasks have the goal to classify nodes in a graph. Usually, we have given a single, large graph with >1000 nodes of which a certain amount of nodes are labeled. **We learn to classify those labeled examples during training and try to generalize to the unlabeled nodes.** A popular example that we will use in this tutorial is the Cora dataset, a citation network among papers. The Cora consists of 2708 scientific publications with links between each other representing the citation of one paper by another. **The task is to classify each publication into one of seven classes.** Each publication is represented by a bag-of-words vector. This means that we have a vector of 1433 elements for each publication, where a 1 at feature indicates that the i-th word of a pre-defined dictionary is in the article. Binary bag-of-words representations are commonly used when we need very simple encodings, and already have an intuition of what words to expect in a network. There exist much better approaches, but we will leave this to the NLP courses to discuss.

Edge-level tasks: Link prediction

In some applications, we might have to predict on an edge-level instead of node-level. **The most common edge-level task in GNN is link prediction.** Link prediction means that given a graph, **we want to predict whether there will be/should be an edge between two nodes or not.** For example, in a social network, this is used by Facebook and co to propose new friends to you. Again, graph level information can be crucial to perform this task. **The output prediction is usually done by performing a similarity metric on the pair of node features, which should be 1 if there should be a link, and otherwise close to 0.**

Graph-level tasks: Graph classification

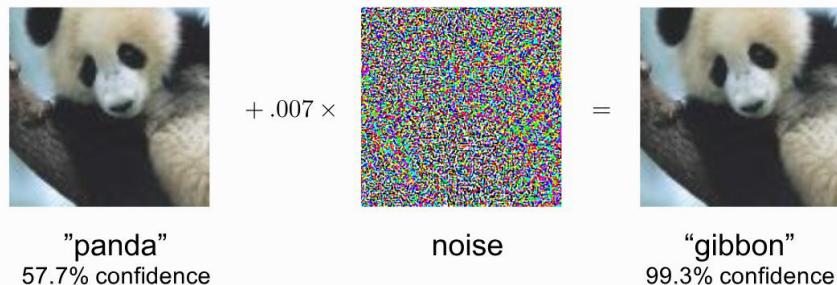
Finally, we will have a closer look at how to apply GNNs to the task of graph classification. **The goal is to classify an entire graph instead of single nodes or edges.** Therefore, we are also given a **dataset of multiple graphs that we need to classify based on some structural graph properties.** The most common task for graph classification is molecular property prediction, in which molecules are represented as graphs. Each atom is linked to a node, and edges in the graph are the bonds between atoms. For example, look at the figure below.



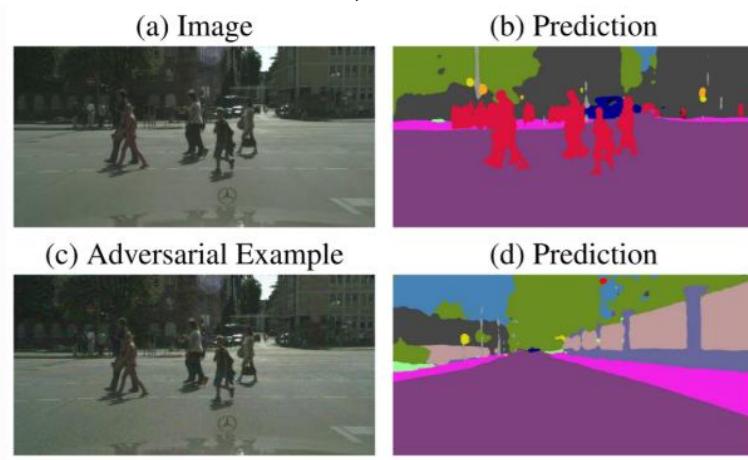
On the left, we have an arbitrary, small molecule with different atoms, whereas the right part of the image shows the graph representation. The atom types are abstracted as node features (e.g. a one-hot vector), and the different bond types are used as edge features. The graph nodes have 7 different labels/atom types, and the binary graph labels represent “their mutagenic effect on a specific gram negative bacterium” (the specific meaning of the labels are not too important here).

Adversarial attacks

Deep Neural Networks are a very powerful tool to recognize patterns in data, and, for example, perform image classification on a human-level. However, we have not tested yet how robust these models actually are. Can we “trick” the model and find failure modes? Can we design images that the networks naturally classify incorrectly? Due to the high classification accuracy on unseen test data, we would expect that this can be difficult. However, in 2014, a research group at Google and NYU showed that deep CNNs can be easily fooled, just by adding some salient but carefully constructed noise to the images. For instance, take a look at the example below:



The image on the left is the original image from ImageNet, and a deep CNN classifies the image correctly as “panda” with a class likelihood of 57%. Nevertheless, if we add a little noise to every pixel of the image, the prediction of the model changes completely. Instead of a panda, our CNN tells us that the image contains a “gibbon” with the confidence of over 99%. For a human, however, these two images look exactly alike, and you cannot distinguish which one has noise added and which doesn’t. While this first seems like a fun game to fool trained networks, it can have a serious impact on the usage of neural networks. More and more deep learning models are used in applications, such as for example autonomous driving. Imagine that someone who gains access to the camera input of the car, could make pedestrians “disappear” for the image understanding network by simply adding some noise to the input as shown below (the figure is taken from [J.H. Metzen et al.](#)). The first row shows the original image with the semantic segmentation output on the right (pedestrians red), while the second row shows the image with small noise and the corresponding segmentation prediction. The pedestrian becomes invisible for the network, and the car would think the road is clear ahead.



White-box adversarial attacks

There have been proposed many possible adversarial attack strategies, which all share the same goal: alternate the data/image input only a little bit to have a great impact on the model's prediction. Specifically, how can we have to change the image so that the model does not recognize it anymore? At the same time, the label of the image should not change, in the sense that a human would still clearly classify it correctly. This is the same objective that the generator network has in the Generative Adversarial Network framework: try to fool another network (discriminator) by changing its input.

Adversarial attacks are usually grouped into “white-box” and “black-box” attacks. **White-box attacks** assume that we **have access to the model parameter** and can, for example, calculate the gradients with respect to the input (similar as in GANs). **Black-box attacks** on the other hand have the harder task of not having any knowledge about the network, and **can only obtain predictions for an image**, but no gradients or the like.

Fast Gradient Sign Method (FGSM)

One of the first attack strategies proposed is Fast Gradient Sign Method (FGSM), developed by [Ian Goodfellow et al.](#) in 2014. Given an image, we create an adversarial example by the following expression:

$$\tilde{x} = x + \epsilon \cdot \text{sign}(\nabla_x J(\theta, x, y))$$

The term $J(\theta, x, y)$ represents the loss of the network for classifying input image x as label y . ϵ is the intensity of the noise, and \tilde{x} the final adversarial example. The equation resembles SGD and is actually nothing else than that. We change the input image x in the direction of **maximizing** the loss $J(\theta, x, y)$. This is exactly the other way round as during training, where we try to minimize the loss. **The sign function and ϵ can be seen as gradient clipping and learning rate specifically.** We only allow our attack to change each pixel value by ϵ . You can also see that the attack can be performed very fast, as it only requires a single forward and backward pass. As expected, the model is fooled on almost every image at least for the top-1 error, and more than half don't have the true label in their top-5. This is a quite significant difference compared to the error rate of 4.3% on the clean images. However, note that the predictions remain semantically similar. FGSM could be adapted to increase the probability of a specific class instead of minimizing the probability of a label, but for those, there are usually better attacks such as the adversarial patch.

Protecting against adversarial attacks

There are many more attack strategies than just FGSM and adversarial patches that we haven't discussed and implemented ourselves here. However, what about the other perspective? What can we do to *protect* a network against adversarial attacks? The sad truth to this is: not much.

White-box attacks require access to the model and its gradient calculation. The easiest way of preventing this is by ensuring safe, private storage of the model and its weights. However, some attacks, called black-box attacks, also work without access to the model's parameters, or white-box attacks can also generalize as we have seen above on our short test on transferability.

So, how could we eventually protect a model? An intuitive approach would be to train/fine-tune a model on such adversarial images, leading to an adversarial training similar to a GAN. During training, we would pretend to be the attacker, and use for example FGSM as an augmentation strategy. However, this usually just ends up in an oscillation of the defending network between

weak spots. Another common trick to increase robustness against adversarial attacks is defensive distillation ([Papernot et al.](#)). Instead of training the model on the dataset labels, we train a secondary model on the softmax predictions of the first one. This way, the loss surface is “smoothed” in the directions an attacker might try to exploit, and it becomes more difficult for the attacker to find adversarial examples. Nevertheless, there hasn’t been found the one, true strategy that works against all possible adversarial attacks.

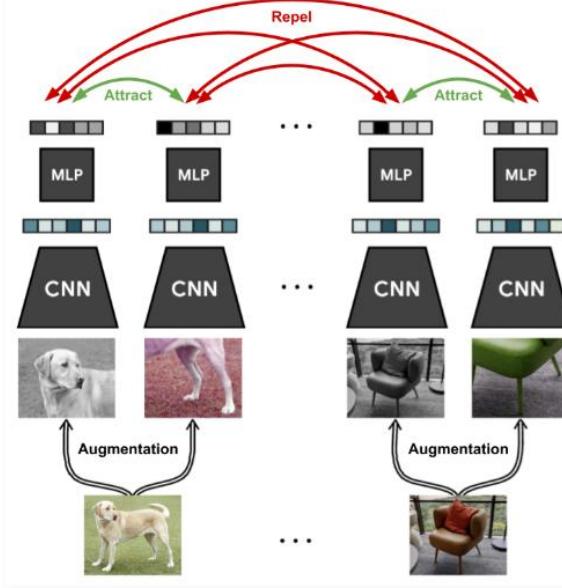
Why are CNNs, or neural networks in general, so vulnerable to adversarial attacks? While there are many possible explanations, the most intuitive is that neural networks don’t know what they don’t know. Even a large dataset represents just a few sparse points in the extremely large space of possible images. A lot of the input space has not been seen by the network during training, and hence, we cannot guarantee that the prediction for those images is any useful. The network instead learns a very good classification on a smaller region, often referred to as manifold, while ignoring the points outside of it. NNs with uncertainty prediction could potentially help to discover what the network does not know. Another possible explanation lies in the activation function. As we know, most CNNs use ReLU-based activation functions. While those have enabled great success in training deep neural networks due to their stable gradient for positive values, they also constitute a possible flaw. The output range of a ReLU neuron can be arbitrarily high. Thus, if we design a patch or the noise in the image to cause a very high value for a single neuron, it can overpower many other features in the network. Thus, although ReLU stabilizes training, it also offers a potential point of attack for adversaries.

Self-Supervised Contrastive Learning with SimCLR

Now, we will take a closer look at **self-supervised contrastive learning**. Self-supervised learning, or also sometimes called unsupervised learning, describes the scenario where we have given input data, but no accompanying labels to train in a classical supervised way. However, this data still contains a lot of information from which we can learn: how are the images different from each other? What patterns are descriptive for certain images? Can we cluster the images? And so on. **Methods for self-supervised learning try to learn as much as possible from the data alone, so it can quickly be finetuned for a specific classification task.** The benefit of self-supervised learning is that a large dataset can often easily be obtained. For instance, if we want to train a vision model on semantic segmentation for autonomous driving, we can collect large amounts of data by simply installing a camera in a car, and driving through a city for an hour. In contrast, **if we would want to do supervised learning, we would have to manually label all those images before training a model.** This is extremely expensive, and would likely take a couple of months to manually label the same amount of data. Further, self-supervised learning can provide an alternative to transfer learning from models pretrained on ImageNet since we could pretrain a model on a specific dataset/situation, e.g. traffic scenarios for autonomous driving.

Within the last two years, a lot of new approaches have been proposed for self-supervised learning, in particular for images, that have resulted in great improvements over supervised models when few labels are available. The subfield that we will focus on in this tutorial is **contrastive learning**.

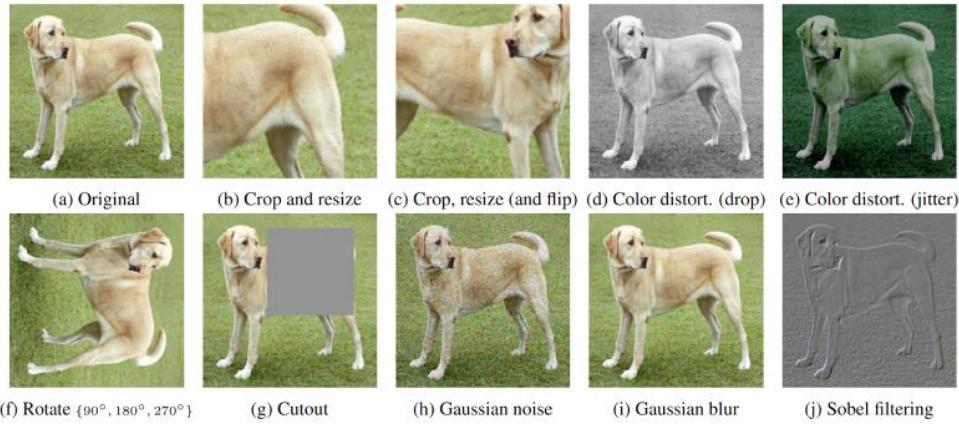
Contrastive learning is motivated by the question mentioned above: **how are images different from each other?** Specifically, **contrastive learning methods train a model to cluster an image and its slightly augmented version in latent space, while the distance to other images should be maximized**. A very recent and simple method for this is [SimCLR](#), which is visualized below:



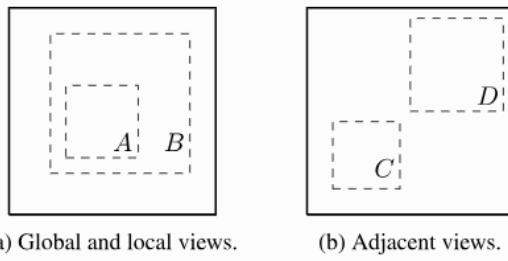
The general setup is that **we are given a dataset of images without any labels, and want to train a model on this data such that it can quickly adapt to any image recognition task afterward**. During each training iteration, we sample a batch of images as usual. **For each image, we create two versions by applying data augmentation techniques like cropping, Gaussian noise, blurring, etc.** An example of such is shown on the left with the image of the dog. On those images, we apply a CNN like ResNet and obtain as output a 1D feature vector on which we apply a small MLP. **The output features of the two augmented images are then trained to be close to each other, while all other images in that batch should be as different as possible.** This way, the model has to learn to recognize the content of the image that remains unchanged under the data augmentations, such as objects which we usually care about in supervised tasks.

SimCLR

We will start our exploration of contrastive learning by discussing the effect of different data augmentation techniques. To allow efficient training, we need to prepare the data loading such that we sample two different, random augmentations for each image in the batch. The easiest way to do this is by creating a transformation that, when being called, applies a set of data augmentations to an image twice. The contrastive learning framework can easily be extended to have more *positive* examples by sampling more than two augmentations of the same image. However, the most efficient training is usually obtained by using only two. Next, we can look at the specific augmentations we want to apply. **The choice of the data augmentation to use is the most crucial hyperparameter in SimCLR** since it directly affects how the latent space is structured, and what patterns might be learned from the data. Let's first take a look at some of the most popular data augmentations.



All of them can be used, but it turns out that two augmentations stand out in their importance: **crop-and-resize, and color distortion**. Interestingly, however, they only lead to strong performance if they have been used together as discussed by [Ting Chen et al.](#) in their SimCLR paper. When performing randomly cropping and resizing, we can distinguish between two situations: (a) cropped image A provides a local view of cropped image B, or (b) cropped images C and D show neighboring views of the same image.



(a) Global and local views.

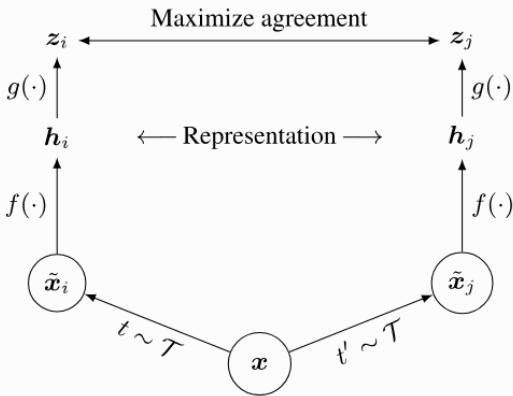
(b) Adjacent views.

While situation (a) requires the model to learn some sort of scale invariance to make crops A and B similar in latent space, situation (b) is more challenging since the model needs to recognize an object beyond its limited view. However, without color distortion, there is a loophole that the model can exploit, namely that **different crops of the same image usually look very similar in color space**. Consider the picture of the dog above. Simply from the color of the fur and the green color tone of the background, you can reason that two patches belong to the same image without actually recognizing the dog in the picture. In this case, the model might end up focusing only on the color histograms of the images, and ignore other more generalizable features. If, however, we distort the colors in the two patches randomly and independently of each other, the model cannot rely on this simple feature anymore. **Hence, by combining random cropping and color distortions, the model can only match two patches by learning generalizable representations.**

SimCLR implementation

Using the data loader pipeline above, we can now implement SimCLR. At each iteration, we get for every image x two differently augmented versions, which we refer to as \tilde{x}_i and \tilde{x}_j . Both of these images are encoded into a one-dimensional feature vector, **between which we want to maximize similarity which minimizes it to all other images in the batch**. The encoder network is split into two parts: a base encoder network $f(\cdot)$, and a projection head $g(\cdot)$. The base network is usually a deep CNN, and is responsible for extracting a representation vector from the augmented data examples. In our experiments, we will use the common ResNet-18 architecture as $f(\cdot)$, and refer to the output as $f(\tilde{x}_i) = h_i$. The projection head $g(\cdot)$ maps the representation h into a space where we apply the contrastive loss, i.e., compare similarities

between vectors. It is often chosen to be a small MLP with non-linearities, and for simplicity, we follow the original SimCLR paper setup by defining it as a two-layer MLP with ReLU activation in the hidden layer. **Note that in the follow-up paper, SimCLRV2, the authors mention that larger/wider MLPs can boost the performance considerably.** This is why we apply an MLP with four times larger hidden dimensions, but deeper MLPs showed to overfit on the given dataset. The general setup is visualized below



After finishing the training with contrastive learning, we will remove the projection head $g(\cdot)$, and use $f(\cdot)$ as a pretrained feature extractor. **The representations z that come out of the projection head $g(\cdot)$ have been shown to perform worse than those of the base network $f(\cdot)$** when finetuning the network for a new task. This is likely because the representations are trained to become invariant to many features like the color that can be important for downstream tasks. Thus, $g(\cdot)$ is only needed for the contrastive learning stage.

Now that the architecture is described, let's take a closer look at how we train the model. As mentioned before, we want to maximize the similarity between the representations of the two augmented versions of the same image, i.e., z_i and z_j in the figure above, while minimizing it to all other examples in the batch. **SimCLR thereby applies the InfoNCE loss**, originally proposed by [Aaron van den Oord et al.](#) for contrastive learning. In short, **the InfoNCE loss compares the similarity of z_i to the similarity of z_j to any other representation in the batch by performing a softmax over the similarity values**. The loss can be formally written as:

$$\ell_{i,j} = -\log \frac{\exp(\text{sim}(z_i, z_j)/\tau)}{\sum_{k=1}^{2N} \mathbb{1}_{[k \neq i]} \exp(\text{sim}(z_i, z_k)/\tau)} = -\text{sim}(z_i, z_j)/\tau + \log \left[\sum_{k=1}^{2N} \mathbb{1}_{[k \neq i]} \exp(\text{sim}(z_i, z_k)/\tau) \right]$$

The function **sim** is a similarity metric, and the hyperparameter τ is called **temperature determining how peaked the distribution is**. Since many similarity metrics are bounded, the temperature parameter allows us to balance the influence of many dissimilar image patches versus one similar patch. The similarity metric that is used in SimCLR is cosine similarity, as defined below:

$$\text{sim}(z_i, z_j) = \frac{z_i^\top \cdot z_j}{\|z_i\| \cdot \|z_j\|}$$

The maximum cosine similarity possible is 1, while the minimum is -1. In general, we will see that the features of two different images will converge to a cosine similarity around zero since the minimum, -1, would require z_i and z_j to be in the exact opposite direction in all feature dimensions, which does not allow for great flexibility.

A common observation in contrastive learning is that **the larger the batch size, the better the models perform**. A larger batch size allows us to compare each image to more negative examples, leading to overall smoother loss gradients. Another thing to note is that contrastive learning benefits a lot from long training.

Logistic Regression

After we have trained our model via contrastive learning, **we can deploy it on downstream tasks and see how well it performs with little data**. A common setup, which also verifies whether the model has learned generalized representations, is to perform Logistic Regression on the features. In other words, we learn a single, linear layer that maps the representations to a class prediction. Since the base network $f(\cdot)$ is not changed during the training process, the model can only perform well if the representations of h describe all features that might be necessary for the task. Further, we do not have to worry too much about overfitting since we have very few parameters that are trained. Hence, we might expect that the model can perform well even with very little data. If very little data is available, it might be beneficial to dynamically encode the images during training so that we can also apply data augmentations.

