Tutorial: How to Build a Deep Learning Framework

Katharina Breininger and Tobias Würfl
Friedrich-Alexander-University Erlangen-Nürnberg, Erlangen, Germany

Introduction

Neural networks, especially convolutional neural networks (CNNs), have had an incredible impact on research in medical imaging and medical signal processing in recent years. Frameworks like TensorFlow, Caffe and PyTorch make it easy to implement network architectures to carry out experiments by simply stacking together operators. This has helped to speed up research immensely - it is easy to try out new ideas and translate insights from other fields. BUT: Never having to understand the technical details of the frameworks and operators deprives researchers of one avenue to potential innovation in the field. Improvements like trainable region-proposals and depth-wise separable convolutions are easier to come up with a thorough understanding the details of the machinery. Many essential advances in DL, such as the ReLU, batch normalization and better initialization strategies, have originated in understanding and improving drawbacks of building blocks in neural networks.
With this tutorial, we aim to support you in understanding what's going in neural network frameworks in detail, and teach you how the most common operators work during inference in a network and how they are adapted by training. This will enable you to implement a broader range of ideas, relying on innovative new operators embedded into neural networks.
As prerequisites we expect some conceptual knowledge about neural networks as acquired in offline or online courses, like the Stanford DL course, or our course (DL_course_videos), as well as basic Python/NumPy programming experience.

- How it works:
  - We provide the necessary math and code skeletons of building blocks
  - You translate this math into code
  - Our unit-tests will give you feedback on the correctness of your implementation
  - At the end, we will put these building blocks together to a working network
- What we don't do:
  - Teach you Python programming
  - Teach you about the fundamentals of machine learning
  - Give a thorough introduction into the subject of deep learning
  - Implement a framework with a focus on performance and efficiency
- Elements in this tutorial
  - Implementing a multilayer perceptron framework
  - Extending this framework with state-of-the-art initialization
  - Adding the basic operators of CNNs

- Including some operators for regularization to the framework

If you have feedback or suggestions for improvement, please contact us at katharina.breininger@fau.de and tobias.wuerfl@fau.de.
Have fun!

```
In [ ]: # minor set-up work
        import numpy  # we will definitely need this

        # automatic reloading
        %load_ext autoreload
        %autoreload 2

        %matplotlib inline
```

The General Idea of the Framework

Almost all tasks in this tutorial will revolve around implementing "layers". All layers are derived from the base class defined in the next cell. Each layer needs to implement the methods `forward` and `backward`. We will use the term "layer" to represent any operator in the network that can be considered as a "unit" during forward and backward pass, e.g., a "fully connected layer", an "activation layer" or a "loss layer".
In `forward(x)`, the forward pass of the layer is computed by applying the respective operation to the input `x`. Furthermore, intermediate results necessary to compute the gradients in the backward pass have to be stored. In `backward(error)`, the layer receives the error passed down from the subsequent layer, updates its parameters accordingly and returns the error with respect to its input.
This way, a simple network for classification can be expressed by a list of layer objects. Given an initial input `x` and a corresponding `label`, the forward pass through the network is computed by subsequently calling `forward` for each layer in the list. The respective output is passed as input to the next layer. The very last layer, the "loss" layer, additionally receives the label to compute the loss. To adapt the weights in each layer, we then go backwards through the list, calling `backward`, backpropagating the error through the network. The network is trained by alternating the forward and backward pass through the network while iterating through the training data.
During test-time, only the forward pass through the network is computed to generate a prediction.

Basic notation and terminology

We will work with the following notation and terminology:

- $\mathbf{X}$ and $\mathbf{x}$ represent the input,
- $\mathbf{W}$ and $\mathbf{w}$ the trainable weights/parameters and
- $\mathbf{Y}$ and $\mathbf{y}$ the output of a layer.
- L represents the loss. Accordingly,
- $E_{\mathbf{Y}} = \frac{\partial L}{\partial \mathbf{Y}}$ is the error passed down from the subsequent layer,
- $E_{\mathbf{W}} = \frac{\partial L}{\partial \mathbf{W}}$ the error with respect to the weights and
- $E_{\mathbf{X}} = \frac{\partial L}{\partial \mathbf{X}}$ is the error with respect to the input.

Note that x and y always have "local" meaning, i.e., with respect to the **current** layer. The y of the previous layer is the x to the next, and vice versa.
Have a look at the class definitions below and make yourself familiar with the concepts before continuing with the next part of the tutorial, the fully connected layer.

```
In [ ]: # %load src/base.py
        def enum(*sequential, **named):
            # Enum definition for backcompatibility
            enums = dict(zip(sequential, range(len(sequential))), **named)
            return type('Enum', (), enums)

        # Enum to encode the which phase a layer is in at the moment.
        Phase = enum('train', 'test', 'validation')

        class BaseLayer:

            def __init__(self):
                self.phase = Phase.train

            def forward(self, x):
                """ Return the result of the forward pass of this layer.
                Save intermediate results necessary to compute the
                gradients in the backward pass.
                """
                raise NotImplementedError('Base class - method is not implemented')

            def backward(self, error):
                """ Update the parameters/weights of this layer (if applicable),
                and return the gradient with respect to the input.
                """
                raise NotImplementedError('Base class - method is not implemented')
```

Fully Connected Layers

Fully connected (FC) layers are the essential building blocks in (multi-layer) perceptrons. Inspired by biological neurons, they are able to represent any connection topology between two layers (without same-layer connections).
Let's have a look at the forward pass: Given an input vector $\mathbf{x} \in \mathbb{R}^n$ to an FC layer, the output y of a single neuron can be described as a weighted sum of the input values plus a bias:

$$y = w_{n+1} + \sum_{j=1}^{n} w_j x_j,$$

where we collect the weights in a vector $\mathbf{w} \in \mathbb{R}^{n+1}$.
This is simply a vector-vector multiplication:

$$y = \begin{pmatrix} w_1 & \dots & w_n & w_{n+1} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \\ 1 \end{pmatrix}$$

By extending $\mathbf{x}$ with an additional "1", we can include the bias directly in the multiplication.
Since we want to have a layer able to generate multiple outputs, we need multiple neurons:
To achieve this, we extend the weight vector to a matrix to allow for an output vector $\mathbf{y} \in \mathbb{R}^m$:

$$\begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix} = \begin{pmatrix} w_{1,1} & \dots & w_{n,1} & w_{n+1,1} \\ \vdots & \ddots & \vdots & \vdots \\ w_{1,m} & \dots & w_{n,m} & w_{n+1,m} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \\ 1 \end{pmatrix}$$

$$\mathbf{y} = \mathbf{W}\mathbf{x}$$

For batch processing, we can accordingly stack multiple input vectors in a matrix $\mathbf{X}$:

$$\mathbf{Y} = \mathbf{W}\mathbf{X}$$

The weight matrix represents the trainable parameters of the FC layer. To be able to update the parameters, we need the gradient of the loss with respect to these weights. Given the error with respect to the output $\mathbf{Y}$ of the current layer $\frac{\partial L}{\partial \mathbf{Y}} = E_{\mathbf{Y}}$, we can compute the gradient with respect to the weights $\frac{\partial L}{\partial \mathbf{W}} = E_{\mathbf{W}}$ using backpropagation, i.e., the chain rule. To backpropagate the error to the previous layer (and then update the weights there), we further need to compute the error with respect to the inputs $\frac{\partial L}{\partial \mathbf{X}} = E_{\mathbf{X}}$.
Using the formula of the fully connected layer $\mathbf{Y} = \mathbf{W}\mathbf{X}$, we can compute the wanted gradients:

$$\begin{aligned} \frac{\partial L}{\partial \mathbf{W}} &= \frac{\partial L}{\partial \mathbf{Y}} \frac{\partial \mathbf{Y}}{\partial \mathbf{W}} \\ &= E_{\mathbf{Y}} \mathbf{X}^T \end{aligned}$$

$$\begin{aligned} \frac{\partial L}{\partial \mathbf{X}} &= \frac{\partial L}{\partial \mathbf{Y}} \frac{\partial \mathbf{Y}}{\partial \mathbf{X}} \\ &= \mathbf{W}^T E_{\mathbf{Y}} \end{aligned}$$

We will use (mini-batch) stochastic gradient descent in this tutorial, so the update rule for the weights is as follows:

$$\mathbf{W}^{t+1} = \mathbf{W}^t - \eta E_{\mathbf{W}^t} \ ,$$

where η is the learning rate and t denotes the iteration.

Implementation task

Now it is your turn: In the next cell, implement the methods `init`, `forward`, `backward`, and `get_gradient_weights` and test the method by running the cell after the next. The method `get_gradient_weights` should return the gradient with respect to the weights and biases of the last backward pass.
Note that input and output, and accordingly the respective errors, are actually transposed compared to the formulas above. This is due to performance reasons and consistency with known frameworks. Make sure to consider this in your implementation.
Furthermore, implement the method `initialize`. For the moment, take the initializer objects as given, we will return to them later. Just make sure to use them with the correct weight shapes to initialize weights and biases. Implement the update of these parameters as part of the backward pass.

```
In [ ]: # %load src/layers/fully_connected.py
        class FullyConnectedLayer(BaseLayer):
            def __init__(self, input_size, output_size, learning_rate):
                """ A fully connected layer.
                    param: input_size (int): dimension n of the input vector
                    param: output_size (int): dimension m of the output vector
                    param: learning_rate (float): the learning rate of this layer
                """
                # TODO: define the neccesary class variables
                pass

            def forward(self, x):
                """ Compute the foward pass through the layer.
                    param: x (np.ndarray): input with shape [b, n] where b is the
                           batch size and n is the input size
                    returns (np.ndarray): result of the forward pass,
                           of shape [b, m] where b is the batch size and m is the
                           output size
                """
                # TODO: Implement forward pass of the fully connected layer
                # Hint: Think about what you need to store during the forward pass
                # to be able to compute
                # the gradients in the backward pass
                pass

            def get_gradient_weights(self):
                """
                    returns (np.ndarray): the gradient with respect to the weights
                            and biases from the last call of backward(...)
                """
                # TODO: Implement
                pass

            def backward(self, error):
```

```
        """ Update the weights of this layer and return the gradient with
        respect to the previous layer.
            param: error (np.ndarray): of shape [b, m] where b is the batch
                   size and m is the output size
            returns (np.ndarray): the gradient w.r.t. the previous layer,
                   of shape [b, n] where b is the batch size and n is the
                   input size
        """
        # TODO: Implement backward pass of the fully connected layer
        # Hint: Be careful about the order of applying the update to the weights
        # and the calculation of
        # the error with respect to the previous layer.
        pass

    def initialize(self, weights_initializer, bias_initializer):
        """ Initializes the weights/bias of this layer with the given
        initializers.
            param: weights_initializer: object providing a method
                   weights_initializer.initialize(weights_shape) which will
                   return initialized weights with the given shape
            param: bias_initializer: object providing a method
                   bias_initializer.initialize(bias_shape) which will return
                   an initialized bias with the given shape
        """
        # TODO: Implement
        pass
```

```
In [ ]: # Running the testsuite
        %run Tests/TestFullyConnected.py
        TestFullyConnected.FullyConnected = FullyConnectedLayer
        unittest.main(argv=['first-arg-is-ignored'], exit=False)
```

Activation Functions

Activation functions play an essential role in neural networks: They introduce non-linearity. In this tutorial, we are going to implement two activation functions: The sigmoid and the rectified linear unit (ReLU).

Sigmoid activation function

Historically, the Sigmoid function has played a big role in the development of neural networks. Given the motivation of biological neurons and their all-or-nothing response, the sigmoid is an obvious choice close to a true step function: It scales the input between 0 and 1, and its gradient exists everywhere. For each element of the input, it is defined as:

$$\text{sig}(x) = \frac{1}{1 + e^{-x}} \ .$$

To be able to backpropagate the error through the network, we need the gradient with respect to the input.

$$\begin{aligned}\frac{\partial \text{sig}(x)}{\partial x} &= \frac{1}{1+e^{-x}}(1-\frac{1}{1+e^{-x}}) \\ &= \text{sig}(x)(1-\text{sig}(x)) \ .\end{aligned}$$

ReLU activation function

While the sigmoid function is still frequently used for example in recurrent networks and as the last layer for binary segmentation/classification, it has been overtaken by the rectified linear unit (ReLU) and its variants in many other setting. The main drawback of the sigmoid function is that its gradient is close to zero everywhere apart from a small region around the origin. This can cause the so-called vanishing gradient problem, meaning that the network will learn very slow or will stop learning completely. The ReLU is much less affected by this problem, as the output is linear for inputs > 0:

$$\text{relu}(x) = \begin{cases} x & \text{if } x > 0, \\ 0 & \text{else.} \end{cases}$$

However, due to the kink at position 0, the function is not continuously differentiable. Instead, we need to compute subgradients in the backward pass:

$$\frac{\partial \text{relu}(x)}{x} = \begin{cases} 1 & \text{if } x > 0, \\ 0 & \text{else.} \end{cases}$$

For both activation functions, we need to apply the chain rule to compute the result of the backward pass:

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial f(x)} \frac{\partial f(x)}{\partial x} \ ,$$

where $f(x)$ stands for any of the two functions.

Implementation task

In the following, implement the `Simoid` and `ReLU` activation functions. Test your implementation by running the cell below.

```
In [ ]: # %load src/layers/activation_functions.py
        class Sigmoid(BaseLayer):

            def forward(self, x):
                """ Return the element-wise sigmoid of the input.
                    param: x (np.ndarray): input to the activation function,
                           of arbitrary shape
                    returns (np.ndarray): element-wise sigmoid(x),
                           of the same shape as x
                """
                # TODO: Implement forward pass of the Sigmoid
                pass
```

```
        def backward(self, error):
            """ Return the gradient with respect to the input.
                param: error (np.ndarray): the gradient passed down from
                       the subsequent layer, of the same shape as x in the
                       forward pass
                returns (np.ndarray): the gradient with respect to the
                       previous layer, of the same shape as error
            """
            # TODO: Implement backward pass of the Sigmoid
            pass


    class ReLU(BaseLayer):

        def forward(self, x):
            """ Return the result of a ReLU activation of the input.
                param: x (np.ndarray): input to the activation function,
                       of arbitrary shape
                returns (np.ndarray): element-wise ReLU(x), of the same shape
                       as x
            """
            # TODO: Implement forward pass of the ReLU
            pass

        def backward(self, error):
            """ Return the gradient with respect to the input.
                param: error (np.ndarray): the gradient passed down from the
                       previous layer, arbitrary shape (same as x)
                returns (np.ndarray): gradient with respect to the input,
                       of the same shape as error
            """
            # TODO: Implement backward pass of the ReLU
            pass
```

```
In [ ]: %run Tests/TestActivationFunctions.py
        TestReLU.ReLU = ReLU
        TestSigmoid.Sigmoid = Sigmoid
        unittest.main(argv=['first-arg-is-ignored'], exit=False)
```

Softmax and Loss Layer

By combining the layers we implemented so far, we can represent a non-linear function of the input. For example, we can compute an output vector with K elements to classify between K classes.

Softmax

The output of this computation is not further restricted. In many cases, however, it is beneficial if a prediction for the targeted classification has the properties of a probability distribution, i.e.,

$$\sum_{k=1}^{K} y_k = 1 \ ,$$
$$y_k \leq 0 \quad \forall k \text{ in } 1, ..., K \ .$$

This makes it for example easier to compare the prediction with the ground truth of the classification task. We can achieve these properties by applying the softmax function as a last activation function. It is defined as:

$$\text{softmax}(x_k) = \frac{\exp(x_k)}{\sum_{j=1}^{K} \exp(x_j)} \ .$$

However, if the activations in $\mathbf{x}$ are high, $\exp(x_k)$ can become very large. This can cause numerical instabilities. To avoid this, the activations can be shifted by the maximum value of $\mathbf{x}$ before applying the softmax:

$$\widetilde{\mathbf{x}} = \mathbf{x} - \max(\mathbf{x}) \ .$$

After the softmax, the predictions of the network have the properties of a probability distribution.

Loss function

To adapt the parameters of the network, we to know how "well" the network performs compared to a given ground truth (or label) - we need a loss function. Then, we can "train" the network by minimizing this loss by iteratively adapting the weights and biases using our training data.
A common loss function is cross entropy. To compute it, we need the ground truth $\mathbf{y}^*$ in "one-hot"-vector encoding. The ground truth is represented as a vector with K elements where only the value that corresponds to the true class is $\neq 0$:

$$\mathbf{y}^* = \begin{pmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{pmatrix}$$

Then, the cross entropy loss for a batch of b samples is defined as:

$$L(\mathbf{Y}^*, \mathbf{Y}) = -\sum_{b} \sum_{k=1}^{K} \ln(y_{b,k}) y^*_{b,k}$$

Combining both

The softmax activation and the cross entropy loss are frequently combined, and sometimes called the "SoftMax loss". Together, their gradient has a simple and elegant form:

$$e_k = y_k - y_k^* \ .$$

for every element of the batch.

Implementation task

Implement the softmax function and the cross entropy loss combined in the class `SoftMaxCrossEntropyLoss`. Since the two functions are combined in `forward`, additionally implement a function `predict` that computes only the softmax of the input. This function can be used during test-time, when we are interested in a prediction for unseen data.

```
In [ ]: # %load src/layers/softmax_crossentropy.py
        class SoftMaxCrossEntropyLoss(BaseLayer):

            def forward(self, x, labels):
                """ Return the cross entropy loss of the input and the labels after
                applying the softmax to the input.
                    param: x (np.ndarray): input, of shape [b, k] where b is the
                           batch size and k is the input size
                    param: labels (np.ndarray): the corresponding labels of the
                           training set in one-hot encoding for the current input,
                           of the same shape as x
                    returns (float): the loss of the current prediction and the label
                """
                # Todo: Implement forward pass
                pass

            def backward(self, labels):
                """ Return the gradient of the SoftMaxCrossEntropy loss with respect
                to the previous layer.
                    param: labels (np.ndarray): (again) the corresponding labels of
                           the training set for the current input, of shape [b, k]
                           where b is the batch size and k is the input size
                    returns (np.ndarray): the error w.r.t. the previous layer,
                           of shape [b, k] where b is the batch size and n is the
                           input size
                """
                # TODO: Implement backward pass
                pass

            def predict(self, x):
                """ Return the softmax of the input.  This can be interpreted as
                probabilistic prediction of the class.
                    param: x (np.ndarray): input with shape [b, k], where b is the
                           batch size and n is the input size
                    returns (np.ndarray): the result softmax(x), of the same shape
                           as x
                """
```

```
            # TODO: Implement softmax
            pass
```

```
In [ ]: %run Tests/TestSoftMaxCrossEntropyLoss.py
        TestSoftMaxCrossEntropyLoss.SoftMaxCrossEntropyLoss = SoftMaxCrossEntropyLoss
        unittest.main(argv=['first-arg-is-ignored'], exit=False)
```

Initialization

Initialization is very critical for non-convex optimization problems, and neural networks are no exception. The most simple strategy is initialization with a constant value, which is frequently used for bias initialization. Generally, bias initialization with a constant of 0 is common, however, with ReLU as activation function, a small positive value is sensible to reduce the risk of "dying ReLUs".
For other weights in FC layers and for weights in convolutional layers that we will look at in a bit, we need a different initialization strategy. If all weights are initialized with the same value, each node would receive the same update and training becomes impossible. One option to break this symmetry is uniform random initialization. Each element of $\mathbf{W}$ is drawn from a uniform distribution with a certain range, commonly $[0, 1]$.
However, even with random initialization, finding the right range for weights is still tricky. If the weights are too small, activations become subsequently smaller when they are passed through the layers. Conversely, if they are too large, the signal grows which each subsequent layer. Both effects hinder effective training.
Glorot and Bengio[1] investigated this problem in more detail and presented a strategy to find the "sweet spot" for weight initialization that keeps the variance of the input and output gradient the same. Given certain assumptions, this can be achieved by drawing the weights from a Gaussian distribution $\mathcal{N}(0, \sigma)$ with zero mean and a standard deviation depending on the number of inputs n_{in} and outputs n_{out} of the layer. He et al.[2] showed that for ReLU activations, an adapted version is required to retain this property:

$$\sigma = \frac{2}{n_{\text{in}}} \ .$$

Implementation task

As the next task, implement the initializers `Const`, `UniformRandom` and `He` that provide the method `initialize` for arbitrary weight shapes. For He initialization, the second dimension of `weight_shape` is assumed to be the number of input nodes. As before, run the cell below to test your implementation.

[1] Glorot X. and Bengio Y. Understanding the difficulty of training deep feedforward neural networks. In Proc. AISTATS, PMLR 9:249-256, 2010.

[2] He K. et al. Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification. In CoRR, abs/1502.01852, 2015.

```
In [ ]: # %load src/layers/initializers.py
        class Initializer:
            """ Base class for initializers. """
            def initialize(self, weight_shape):
                """ Return weights initialized according to the subclass definition.
```

```
            Required to work for arbitrary weight shapes.
            Base class.
        """

        # Raises an exception in base class.
        raise NotImplementedError('Method is not implemented')


class Const(Initializer):

    def __init__(self, value):
        """ Create a constant initializer.
            params: value (float): constant that is used for initialization
                    of weights
        """
        # TODO: Implement
        pass

    def initialize(self, weight_shape):
        """ Return a new array of weights initialized with a constant value
        provided by self.value.
            param: weight_shape: shape of the new array
            returns (np.ndarray): array of the given shape
        """
        # TODO: Implement
        pass

class UniformRandom(Initializer):

    def initialize(self, weight_shape):
        """ Return a new array of weights initialized by drawing from a
        uniform distribution with range [0, 1].
            param: weight_shape: shape of new array
            returns (np.ndarray): array of the given shape
        """
        # TODO: Implement
        pass


class He(Initializer):

    def initialize(self, weight_shape):
        """ Return a new array of weights initialized according to
        He et al.: Delving Deep into Rectifiers.
            param: weight_shape: shape of the np.array to be returned,
                    the second dimension is assumed to be the number of
                    input nodes
            returns (np.ndarray): array of the given shape
```

```
        """
        # TODO: Implement
        pass
```

```
In [ ]: %run Tests/TestInitializers.py
        TestInitializers.Const = Const
        TestInitializers.Uniform = UniformRandom
        TestInitializers.He = He
        unittest.main(argv=['first-arg-is-ignored'], exit=False)
```

Convolutional layers

Convolutional layers are without doubt one of the key elements of the success of neural networks in recent years. The main idea is simple: Convolution with trainable filters. They allow to learn which features are important for a given task in a data driven manner. One of their big advantages is that they inherently consider the spatial layout of the data. The animation below shows an example of a 2-D convolution of a padded input (blue) with a 3×3 filter kernel that generates the output in green:
Source: https://github.com/vdumoulin/conv_arithmetic
In this tutorial, we will implement a 2-D convolutional layer that is fully connected in the depth/channel direction. Accordingly, given an input with C channels, each filter has a shape of $M \times N \times C$, where M and N describe the spacial dimensions of the filter. The number of channels of the output depends on the number of filters S in the convolutional layer.
In the example above, the input has $C = 3$ channels and the convolutional layer has $S = 2$ filters fully connected in depth direction. Accordingly, the output has two channels.

Forward pass in a Conv layer:

We can compute the forward pass in multiple ways:

Matrix multiplication Convolution is a linear operation that can be expressed as a matrix multiplication. A downside of this is that a lot of data redundancy is necessary. However, highly efficient algorithms for matrix multiplication can be used.

Convolution The forward pass of a *convolutional* layer can of course also be straight forwardly be implemented as a convolution.

Cross-correlation Cross-correlation is simply a convolution without a flipped filter. For filters that are initialized randomly, we are free to use cross-correlation instead of convolution in the forward pass. We will see that it saves us a bit of kernel flipping in the backward pass.
In all cases, the bias in a convolutional layer is an element-wise addition of a scalar value for each output channel.

Backward pass in a Conv layer:

In the backward pass, we need to compute the gradient with respect to the weights of the convolutional kernel, the bias and the input, given the backpropagated error tensor E_Y.

Matrix multiplication Like in the forward pass, we can implement the backward pass by reusing the formulas from the fully connected layer if we express the convolution as a matrix multiplication.

Convolution/cross-correlation We may want to have a detailed look at the animation above, pick up pen and paper and track which pixels of the input/weight and correspondingly which pixels of the error contribute to respective gradient. For the gradient with respect to the input, we can then see that we need flipped kernel weights in the spacial dimensions (width and height). Alternatively, if we used convolution in the forward pass, we can now apply cross-correlation, and vice versa.
Next, let's have a look at the channels: If we have S kernels in the forward pass, and the input has C channels, we obviously need to re-arrange the weights to C kernels with S channels for the backward pass.
In the animation above shows that channel c of E_X depends only on the channel c of the kernel weights. You can further see how the channels of the kernels can be recombined to compute the gradient with respect to the input.
For the gradient with respect to the weights, you can observe that a correlation operation is necessary: First, the input has to be padded with half the kernel width. Then, each channel of the input has to be correlated with the channel s of E_Y to yield the gradient for kernel s. We have to compute

$$\frac{\partial L}{\partial W_{c,s}} = X_c \star E_{Y_s}$$

for c in $\{1, ..., C\}$ to stack together W_s:
If convolution was used in the forward pass, the result of this correlation represents the flipped gradient, so it has to be flipped back before an update. If correlation was used instead, we save this flipping operation. To really understand this, you may want to grab pen and paper again.
The gradient with respect to the bias can be computed by simply summing over the respective channel.
Like in the fully connected layer, the gradient for the full mini-batch are the sum of the gradient of the elements of the batch.

Stride

Source: https://github.com/vdumoulin/conv_arithmetic
A strided convolution can be used to downsample the input. From a mathematical perspective, this can be expressed as a convolution followed by subsampling. Similarly, in the backward pass, E_Y is first upsampled (introducing zeros), and then processed as before.

Padding

In this tutorial, we will restrict the padding strategy to "same" padding, meaning the input will be padded with zeros such that output after the convolution has the same size as the original input.

Reshaping

Convolutional layers inherently expect the input to have a certain spatial layout with possibly arbitrary size, which is different to FC layers that expect a vector of fixed size. There are two common ways to make these operations interoperable:

- Flatten the input before passing it to an FC layer
- Have the convolutional layers reshape the input to the correct spatial layout

Here, we will implement the first option. To this end, a FlattenLayer is introduced with the sole purpose of reshaping the input to be compatible with FC layers. As no computation is involved, the backward pass simply consists of reversing the reshaping.

Implementation task

In the following, implement the classes `FlattenLayer` and `ConvolutionalLayer` as described above. The necessary parameters are further described in the method documentation.
Note: If you use 3D convolution/correlation (which makes sense from an implementation perspective), keep in mind that you potentially need to compensate for "unnecessary" flipping in the channel dimension in your implementation. Check your implementation by running the unit tests in the subsequent cell.

```
In [ ]: # %load src/layers/conv.py
        class FlattenLayer(BaseLayer):
            def __init__(self):
                # TODO: define the necessary class variables
                pass

            def forward(self, x):
                """ Return a flattened version of the input.
                    param: x (np.ndarray): input, of shape [b, n_channels, p, q]
                           where b is the batch size, n_channels is the number
                           of channels and p x q is the image size
                    returns (np.ndarray): a flattened representation of x
                           of shape [b, v] where b is the batch size and v is the
                           output size = n_channels * p * q
                """
                # TODO: Implement flattening of the image
                pass

            def backward(self, error):
                """ Return the gradient with respect to the input.
                    param: error (np.ndarray): the gradient passed down from the
                           subsequent layer, of shape [b, m], where b is the batch
                           size and m is the output size with m = n_channels * p * q
                           from the forward pass
                    returns (np.ndarray): the error with restored dimensions from the
                           forward pass, i.e. with shape [b, n_channels, p, q] where
                           b is the batch size, n_channels is the number of channels
                           and p x q is the image size
                """
                # TODO: Restore the image dimensions
                pass
```

```python
class ConvolutionalLayer(BaseLayer):

    def __init__(self, stride, kernel_shape, n_kernels, learning_rate):
        """
            param: stride: tuple in the form of (np, nq) which denote the
                   subsampling factor of the convolution operation in the
                   spatial dimensions
            param: kernel_shape: integer tuple in the form of (n_channels, m, n)
                   where n_channels is the number of input channels and m x n is
                   the size of the filter kernels
            param: n_kernels (int): number of kernels and therefore the number
                   of output channels
            param: learning_rate (float): learning rate of this layer
        """
        # TODO: define the neccesary class variables
        pass

    def forward(self, x):
        """ Return the result of the forward pass of the convolutional layer.
            param: x(np.ndarray): input, of shape [b, n_channels, p, q],
                   where b is the batch size, n_channels is the number of
                   input channels and p x q is the image size
            returns (np.ndarray): result of the forward pass,
                   of shape (b, n_kernels, p', q') where b is the batch size,
                   n_kernels is the number of kernels in this layer and
                   p' x q' is the output image size (which depends on the stride)
        """
        # TODO: Implement forward pass of the convolutional layer
        pass

    def backward(self, error):
        """ Update the weights of this layer and return the gradient with
        respect to the input.
            param: error (np.ndarray): of shape (b, n_kernels, p', q')
                   where b is the batch size, n_kernels is the number of kernels
                   and p' x q' is the spacial error size (depends on the stride)
            returns (np.ndarray): the gradient with respect to the input,
                   of shape (b, n_channels, p, q) where b is the batch size,
                   n_channels is the number of input channels to this layer and
                   p x q is the image size.
        """
        # TODO: Implement backward pass of the convolutional layer
        pass

    def get_gradient_weights(self):
        """ Returns the gradient with respect to the weights from the last call
        of backward()
```

```
        """
        # TODO: Implement
        pass

    def get_gradient_bias(self):
        """ Returns the gradient with respect to the bias from the last call
        of backward()
        """
        # TODO: Implement
        pass

    def initialize(self, weights_initializer, bias_initializer):
        """ Initializes the weights/bias of this layer with the given initializers.
            param: weights_initializer: object providing a method
                   weights_initializer.initialize(weights_shape)
                   which will return initialized weights with the given shape
            param: bias_initializer: object providing a method
                   bias_initializer.initialize(bias_shape)
                   which will return an initialized bias with the given shape
        """
        # TODO: Implement. To make sure that He initialization works as intended,
        # make sure the second dimension
        # of weights_shape contains the number of input nodes that can be computed
        # as n_in = n_channels * m * n
        # and reshape the weights to the correct shape afterwards.
        pass
```

```
In [ ]: %run Tests/TestConv.py
        TestConv.Conv = ConvolutionalLayer
        TestConv.FullyConnected = FullyConnectedLayer
        TestConv.He = He
        TestConv.Constant = Const
        TestConv.Flatten = FlattenLayer
        unittest.main(argv=['first-arg-is-ignored'], exit=False)
```

Pooling Layers

As alternative to striding in a convolutional layer, specific pooling layers can be used to downsample the data and condense spacial information. We will look at max pooling as one example. In the forward pass, the output for each pixel is the maximum value in a neighborhood of the corresponding input pixel, calculated separately for every channel. The downsampling is again achieved by using a stride > 1.
Source: https://github.com/vdumoulin/conv_arithmetic
The above example shows maxpooling with a neighborhood of 3×3 and a stride of $[1, 1]$.
The maximum operation can be thought of as an on/off switch for the backpropagation of the gradient for each pixel. We therefore need to store the location of the maximum value in the forward pass. Since the layer has no trainable parameters, we only need to compute the gradient with respect to the input. In the backward pass, the subgradient is given by the colloquial rule "the

winner takes it all". The error is routed only towards the maximum locations; for all other input pixels, the gradient is zero. If the stride is smaller than the neighborhood, the routed gradients for the respective pixels are summed up.

Implementation task

In the following, implement the class `MaxPoolLayer`. Check your implementation as usual by running the unittests in the cell below the implementation.

In [ ]:
```
# %load src/layers/pooling
class MaxPoolLayer(BaseLayer):

    def __init__(self, neighborhood=(2, 2), stride=(2, 2)):
        """ Max pooling layer.
           param: neighborhood: tuple with shape (sp, sq) which denote the
                  kernel size of the pooling operation in the spatial dimensions
           param: stride: tuple with shape (np, nq) which denote the subsampling
                  factor of the pooling operation in
           the spacial dimensions
        """
        # TODO: define necessary class variables
        pass

    def forward(self, x):
        """ Return the result of maxpooling on the input.
            param: x (np.ndarray) with shape (b, n_channels, p, q) where b is the
                   batch size, n_channels is the number of input channels and p x q
                   is the image size
            returns (np.ndarray): the result of max pooling,
                   of shape (b, n_channels, p', q') where b is the batch size,
                   n_channels is the number of input channels and
                   p' x q' is the new image size reduced by the stride.
        """
        # TODO: Implement forward pass of max pooling
        pass

    def backward(self, error):
        """ Return the gradient with respect to the previous layer.
            param: error(np.ndarray): the gradient passed own from the subsequent
                   layer, of shape [b, n_channels, p', q'] where b is the batch size,
                   n_channels is the number of channels and
                   p' x q' is the image size reduced by the stride
            returns (np.ndarray): the gradient w.r.t. the previous layer,
                   of shape [b, n_channels, p, q] where b is the batch size,
                   n_channels is the number of input channels to this layer and
                   p x q is the image size prior to downsampling.
        """
        # TODO: Implement backward pass of max pooling
```

```
                pass
```

```
In [ ]: %run Tests/TestMaxPoolLayer.py
        TestMaxPooling.MaxPooling = MaxPoolLayer
        TestMaxPooling.FullyConnected = FullyConnectedLayer
        TestMaxPooling.Flatten = FlattenLayer
        unittest.main(argv=['first-arg-is-ignored'], exit=False)
```

Dropout

Most successful deep learning models use some regularization techniques intended to decrease the gap between training and test accuracy. The goal is to bias the model towards a model with lower training accuracy but better generalization capability. One prominent technique is dropout. It was for example used in the famous AlexNet network. The idea of this technique is to break dependencies between features by setting random activations to zero during training. This is typically done with a Bernoulli distribution: In each training iteration, the probability for a certain activation to "drop out" is $1 - p$. The application of dropout shifts the mean of the activations because many elements are set to zero during training. At test time, when no element are dropped out, the mean is different, which can decrease performance. To combat this the "training mean" can be restored by multiplying all activations with p at test time.

Inverted dropout

The multiplication at test time can be avoiding by rewriting dropout behaviour during training. This means that the dropout layer can actually be skipped completely during test time, allowing for faster inference. To this end, the activations are multiplied by $\frac{1}{p}$ after applying the stochastic function during training. This way, the mean is not changed by the layer and no operation needs to be performed during test time.

Implementation task

In the following, implement the `DropOut` layer. Both "normal" and inverted dropout are valid implementations. As usual, check your implementation by running the unittests. Note that dropout operates on each element of the input vector independently.

```
In [ ]: # %load src/layers/dropout
        class DropOut(BaseLayer):

            def __init__(self, probability):
                """ DropOut Layer.
                    param: probability: probability of each individual activation to
                    be set to zero, in range [0, 1]
                """
                # TODO: Implement initialization

                pass

            def forward(self, x):
                """ Forward pass through the layer: Set activations of the input
```

```
            randomly to zero.
                param: x (np.ndarray): input
                returns (np.ndarray): a new array of the same shape as x,
                        after dropping random elements
            """
            # TODO: Implement forward pass of the Dropout layer
            # Hint: Make sure to treat training and test phase accordingly.
            pass

        def backward(self, error):
            """ Backward pass through the layer: Return the gradient with
            respect to the input.
                param: error (np.ndarray): error passed down from the subsequent
                        layer, of the same shape as the output of the forward pass
                returns (np.ndarray):  gradient with respect to the input,
                        of the same shape as error
            """
            # TODO: Implement backward pass of the Dropout layer
            pass
```

```
In [ ]: %run Tests/TestDropout.py
        TestDropout.DropOut = DropOut
        TestDropout.Phase = Phase
        unittest.main(argv=['first-arg-is-ignored'], exit=False)
```

LeNet

As the last part of this tutorial, we use our developed operators to construct a simple neural network inspired by the traditional LeNet architecture:
Source: LeCun et al, 1998.[3]
Use two convolutional layers with 5×5 kernels and 6 respectively 10 channels. Each convolution is followed by a ReLU unit and max pooling of with a neighborhood and stride of 2 in each dimension. The top of the network is formed by three FC layers with ReLU activations producing outputs of dimensionality 120, 84 and subsequently the number of categories. Finally, use the SoftMaxCrossEntropyLoss as loss layer.
First, have a look at the class `NeuralNetwork`, that provides the basic framework in which you can use the different layers and stack them together to a functioning network. You don't need to adapt this class, but you can use it to implement the LeNet architecture. You may also want to refer back to the Section **??** in the beginning.

Implementation task

Next, implement the LeNet architecture in the `build` function and train your network in with the script provided below.
Experiment for example with the activation function and DropOut, tune the learning rate or look at the effect of initialization. Feel free to add your own evaluations and plots. You can get the full test data of the MNIST data object by calling `net.data_layer.get_test_set`.

[3] LeCun Y., Bottou L., Bengio Y. and Haffner P. Gradient-based Learning Applied to Document Recognition. In Proc. IEEE, 1989.

In [ ]:
```
# %load src/network.py

# Nothing to do in this cell: Just make yourself familiar with the NeuralNetwork class
class NeuralNetwork:
    def __init__(self, weights_initializer, bias_initializer):
        # list which will contain the loss after training
        self.loss = []
        self.data_layer = None   # the layer providing data
        self.loss_layer = None   # the layer calculating the loss & prediction
        self.layers = []
        self.weights_initializer = weights_initializer
        self.bias_initializer = bias_initializer
        self.label_tensor = None # the labels of the current iteration

    def append_fixed_layer(self, layer):
        """ Add a non-trainable layer to the network. """
        self.layers.append(layer)

    def append_trainable_layer(self, layer):
        """ Add a new layer with trainable parameters to the network.
        Initialize the parameters of the network using the object's
        initializers for weights and bias.
        """
        layer.initialize(self.weights_initializer, self.bias_initializer)
        self.layers.append(layer)

    def forward(self):
        """ Compute the forward pass through the network. """
        # fetch some training data
        input_tensor, self.label_tensor = self.data_layer.forward()
        # defer iterating through the network
        activation_tensor = self.__forward_input(input_tensor)
        # calculate the loss of the network using the final loss layer
        return self.loss_layer.forward(activation_tensor, self.label_tensor)

    def __forward_input(self, input_tensor):
        """ Compute the forward pass through the network,
            stopping before the loss layer.
            param: input_tensor (np.ndarray): input to the network
            returns: activation of the last "regular" layer
        """
        activation_tensor = input_tensor
        # pass the input up the network
        for layer in self.layers:
            activation_tensor = layer.forward(activation_tensor)
        # return the activation of the last layer
        return activation_tensor
```

```
        def backward(self):
            """ Perform the backward pass during training. """
            error_tensor = self.loss_layer.backward(self.label_tensor)
            # pass back the error recursively
            for layer in reversed(self.layers):
                error_tensor = layer.backward(error_tensor)

        def train(self, iterations):
            """ Train the network for a fixed number of steps.
                param: iterations (int): number of iterations for training
            """
            for layer in self.layers:
                layer.phase = Phase.train  # Make sure phase "train" for all layers
            for i in range(iterations):
                loss = self.forward()  # go up the network
                self.loss.append(loss)  # save the loss
                self.backward()  # and down again
                print('.', end='')


        def test(self, input_tensor):
            """ Apply the (trained) network to input data to generate a prediction.
                param: input_tensor (nd.nparray): input (image or vector)
                returns (np.ndarray): prediction by the network
            """
            for layer in self.layers:
                layer.phase = Phase.test  # Make sure phase "test" for all layers
            activation_tensor = self.__forward_input(input_tensor)
            return self.loss_layer.predict(activation_tensor)
```

In [ ]:
```
def build():
    # returns: a neural network architecture built according to the
    # provided specification

    net = NeuralNetwork(He(), Const(0.1))
    learning_rate = 0.001
    categories = 10  # MNIST, numbers 0-9

    # TODO: Implement the architecture by adding layers to net

    return net
```

In [ ]:
```
import matplotlib
import numpy as np
import matplotlib.pyplot as plt

net = build()
```

```
from Tests import Helpers
net.data_layer = Helpers.MNISTData(20)
n_iters = 100
net.train(n_iters)

plt.plot(range(n_iters), net.loss)
```

In [ ]:
```
# Perform the prediction for a random test sample from the dataset:
x, l = net.data_layer.get_random_test_sample()
plt.imshow(x[:28*28].reshape(28, 28), cmap='gray')

print(x.shape)
print('Prediction with highest output: {}'.format(np.argmax(net.test(x))))
print('Ground truth: {}'.format(np.argmax(l)))
```

Summary and Outlook

In this tutorial, we implemented some of the most common building blocks of neural networks, including fully connected layers, activation functions, convolutional layers and regularization operators. Finally, we combined these operators to working network.
We covered only a small subset of elements that are relevant for neural networks. We encourage you to play with other operators, for example batch normalization[4], alternative activation functions, initialization strategies or recurrent units. You may also refactor the framework to experiment with different optimizers, like SGD with momentum, Adam or AdaGrad, or extend the framework to allow for weight decay.
We hope you enjoyed this tutorial and gained a deeper understanding of neural network operators and frameworks. Have fun on your journey further into deep learning and neural networks!

[4] Ioffe S., Szegedy C. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In Proc. ICML, 2015.