Cifar10 Classification

Until now, we have implemented several pieces of a deep learning pipeline and even trained a two-layer neural network, but all the hyperparameters were already set to some values yielding resonable results. In real-life problems, however, much of the work in a deep learning project will be geared towards finding the best hyperparameters for a certain problem. In this notebook we will explore some good practices for network debugging and hyperparameters search, as well as extending our previously binary classification neural network to a multi-class one.

Let's go!

In [1]:
```
# Some lengthy setup.
import matplotlib.pyplot as plt
import numpy as np
import os

from exercise_code.networks.layer import (
    Sigmoid,
    Relu,
    LeakyRelu,
    Tanh,
)
from exercise_code.data import (
    DataLoader,
    ImageFolderDataset,
    RescaleTransform,
    NormalizeTransform,
    FlattenTransform,
    ComposeTransform,
)
from exercise_code.data.image_folder_dataset import RandomHorizontalFlip
from exercise_code.networks import (
    ClassificationNet,
    BCE,
    CrossEntropyFromLogits
)

%load_ext autoreload
%autoreload 2
%matplotlib inline

plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'
```

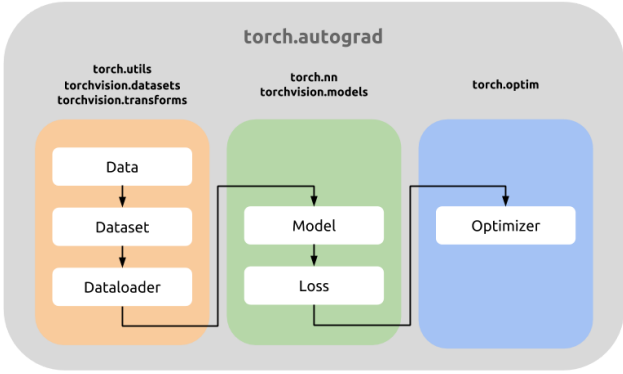
1. Quick recap (and some new things)

Until now, in the previous exercises, we focused on building and understanding all the necessary modules for training a simple model. We followed the Pytorch implementations closely, as this is the framework we will use later and we want you to have a smoother transition to its APIs.

In the figure below you can see the main components in Pytorch. Before starting the actual exercise, we begin with a quick recap of **our implementation** of these components.

Task: Check Code

Everything is already implemented for this part, but we **strongly** encourage you to check out the respective source files in order to have a better understanding.



1.1 Dataset and Dataloader

Data preparation represents an important part of deep learning projects. The data comes from different sources and in different formats and is prepared differently from application to application. One part, however, is clear: because entire datasets are usually too large for us to handle at once, we train our models on smaller batches of data.

The goal of the `Dataset` class is to encapsulate all the 'dirty' data processing: loading and cleaning the data, storing features (or names of files where features can be found) and labels, as well as providing the means for accessing individual (transformed) items of the data using the `__getitem__()` function and an index. You already implemented an `ImageFolderDataset` (in `exercise_code/data/image_folder_dataset.py`) class in Exercise 3. We we will reuse this class here.

For processing the data, you implemented several transforms in Exercise 3 (`RescaleTransform`, `NormalizeTransform`, `ComposeTransform`). In this exercise we are working with images, which are multidimensional arrays, but we are using simple feedforward neural network which takes a one dimensional array as an input, so it is necessary to reshape the images before feeding them into the model.

Task: Check Code

Please check the implementation of the reshape operation in the `FlattenTransform` class, which can be found in `../exercise_06/exercise_code/data/image_folder_dataset.py`.

In [2]:
```
download_url = "https://cdn3.vision.in.tum.de/~dl4cv/cifar10.zip"
i2dl_exercises_path = os.path.dirname(os.path.abspath(os.getcwd()))
cifar_root = os.path.join(i2dl_exercises_path, "datasets", "cifar10")

# Use the Cifar10 mean and standard deviation computed in Exercise 3.
cifar_mean = np.array([0.49191375, 0.48235852, 0.44673872])
cifar_std  = np.array([0.24706447, 0.24346213, 0.26147554])

# Define all the transforms we will apply on the images when
# retrieving them.
rescale_transform = RescaleTransform()
normalize_transform = NormalizeTransform(
    mean=cifar_mean,
    std=cifar_std
)
flatten_transform = FlattenTransform()
compose_transform = ComposeTransform([rescale_transform,
                                      normalize_transform,
                                      flatten_transform])

# Create a train, validation and test dataset.
datasets = {}
for mode in ['train', 'val', 'test']:
    crt_dataset = ImageFolderDataset(
        mode=mode,
        root=cifar_root,
        download_url=download_url,
        transform=compose_transform,
        split={'train': 0.6, 'val': 0.2, 'test': 0.2}
    )
    datasets[mode] = crt_dataset
```

Then, based on this `Dataset` object, we can construct a `Dataloader` object which samples a random mini-batch of data at once.

In [3]:
```
# Create a dataloader for each split.
dataloaders = {}
for mode in ['train', 'val', 'test']:
    crt_dataloader = DataLoader(
        dataset=datasets[mode],
        batch_size=256,
        shuffle=True,
        drop_last=True,
    )
    dataloaders[mode] = crt_dataloader
```

Because the `Dataloader` has the `__iter__()` method, we can simply iterate through the batches it produces, like this:

```
for batch in dataloader['train']:
    do_something(batch)
```

1.2 Data Augmentation

After the above preprocessing steps, our data is in a good shape and ready to be fed into our network. As explained in the chapter above, we used the transformation functions `RescaleTransform`, `NormalizeTransform` and `FlattenTransform` to achieve this shape. These are the general steps that you need to perform on the data before we can even start the training. Of course, all these steps have to be applied to all three splits of our dataset (train, val and test split). So in other words, preprocessing involves preparing the data before they are used in training and inference.

Besides these basic transformations, there are many other transformation methods that you can apply to the images. For example, you can **flip the images horizontally** or **blur the image** and use these new images to enlarge your dataset. This idea is called Data Augmentation and it involves methods that alter the training images to generate a synthetic dataset that is larger than your original dataset and will hopefully improve the performance of your model. The purpose here is different than in the data preprocessing steps and there is one big difference between data augmentation and data preprocessing: The transformation methods to enlarge your dataset should only be applied to the training data. The validation and test data are not affected by these methods.

Task: Check Code

The choice of transformation methods to use for data augmentation can be seen as a hyperparameter of your model and you can try to include these to enlarge your training data and obtain better results for your model. In `exercise_code/data/image_folder_dataset.py` we implemented the function `RandomHorizontalFlip` for you, which is randomly flipping an image. Check out the implementation.

Later, we will apply some hyperparameter tuning and in order to improve your model's accuracy, you could try to include some data augmentation methods. Fell free to play around and maybe also implement some other methods as for example Gaussian Blur or Rotation.

Let us quickly check out the `RandomHorizontalFlip` method with an image of the Cifar10 dataset in the following cell.

there is blank to fill in class MyownNetwork your first fully owned network !

In [4]:
```
#Load the data in a dataset without any transformation
dataset = ImageFolderDataset(
        mode=mode,
        root=cifar_root,
        download_url=download_url,
        split={'train': 0.6, 'val': 0.2, 'test': 0.2},
    )

#Retrieve an image from the dataset and flip it
image = dataset[1]['image']
transform = RandomHorizontalFlip(1)
image_flipped = transform(image)

#Show the two images
plt.figure(figsize = (2,2))
plt.subplot(1, 2, 1)
plt.imshow(image.astype('uint8'))
plt.axis('off')
plt.subplot(1, 2, 2)
plt.imshow(image_flipped.astype('uint8'))
plt.axis('off')
plt.title("Left: Original Image, Right: Flipped image")
plt.show()
```

1.3 Layers

Now, that the data is prepared, we can discuss the model in which we are feeding the data. In our case the model will be a neural network.

In Exercise 5, you implemented a simple 2-layer neural network that had a hidden size as a parameter:

$$\hat{y} = \sigma(\sigma(xW_1 + b_1)W_2 + b_2)$$

where $\sigma(x)$ was the sigmoid function, x was the input, W_1, W_2 the weight matrices and b_1, b_2 the biases for the two layers.

This is how we used this network:

In [5]:
```
input_size = datasets['train'][0]['image'].shape[0]
model = ClassificationNet(input_size=input_size,
                          hidden_size=512)
```

Note that we updated the `ClassificationNet` from the previous exercise, so that now you can customize more: the number of outputs, the choice of activation function, the hidden size etc. We encourage you to check out the implementation in `exercise_code/networks/classification_net.py`

In [6]:
```
num_layer = 2
reg = 0.1

model = ClassificationNet(activation=Sigmoid(),
                          num_layer=num_layer,
                          reg=reg,
                          num_classes=10)
```

Then, the forward and backward passes through the model were simply:

```
# X is a batch of training features
# X.shape = (batch_size, features_size)
y_out = model.forward(X)

# dout is the gradient of the loss function w.r.t the output of the network.
# dout.shape = (batch_size, )
model.backward(dout)
```


Just as the learning rate or the number of iterations we want to train for, the number of hidden layers and the number of units in each hidden layer are also hyperparameters. In this notebook you will play with networks of different sizes and will see the impact that the network capacity has.

Before we move on to the loss functions, we want to have a look at the activation functions. The choice of an activation function can have a huge impact on the performance of the network that you are designing. So far, you have implemented the `Sigmoid` and the `Relu` activation function in Exercise 5.

Task: Check Code

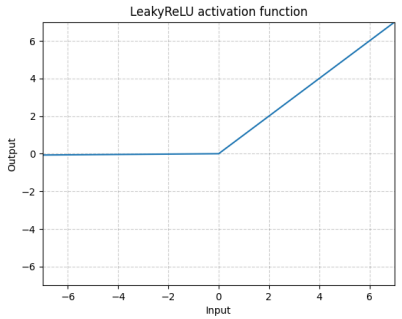
Take a look at the `Sigmoid` and the `Relu` class in `exercise_code/networks/layer.py` and the implementation of the respective forward and backward pass. Make sure to understand why we use **element-wise product** instead of dot product in the backward pass of the `Sigmoid` class to compute the gradient dx. That will be helpful for your later implementation of other activation functions.

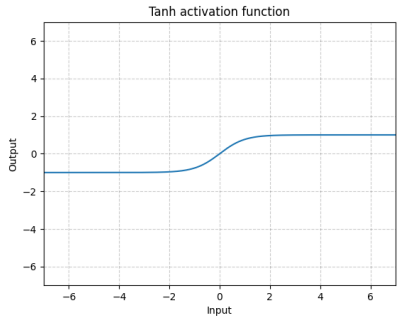
Note: The `cache` variable is used to store information from the forward pass and then pass this information in the backward pass to make use of it there. The implementation of both classes show that this variable can be used differently - depending on what information is needed in the backward pass.

Now, we want to have a look at two other, very common activation functions that you have already met in the lecture: Leaky ReLU activation function and Tanh activation function.

Leaky Relus are one attempt to fix the "dying ReLU" problem. Instead of the function being zero when $x < 0$, a leaky ReLU has a small negative slope (for example, 0.01). That is, the function computes $f(x) = \mathbb{1}(x < 0)(\alpha x) + \mathbb{1}(x >= 0)(x)$ where α is a small constant. Some people report success with this form of activation function, but the results are not always consistent.

The **tanh non-linearity** squashes a real-valued number to the range [-1, 1]. Like the sigmoid neuron, its activations saturate, but unlike the sigmoid neuron its output is zero-centered. Therefore, in practice the tanh non-linearity is always preferred to the sigmoid non-linearity. Also note that the tanh neuron is simply a scaled sigmoid neuron, in particular the following holds: $\tanh(x) = 2 \cdot \sigma(2x) - 1$.





Task: Implement Activation Layers

Now, it is your turn to implement the `LeakyRelu` and the `Tanh` class in `exercise_code/networks/layer.py` by completing the `forward` and the `backward` functions. You can test your implementation in the following two cells.

Note: Always remember to return a cache in `forward` for later backpropagation in `backward`. As we have seen above, the `cache` variable can be used differently for two activation functions.

Use this cell to test your implementation of the `LeakyRelu` class:

In [35]:
```
from exercise_code.tests.layer_tests import *
print(LeakyReluTestWrapper()())
```

```
LeakyReluForwardTest passed.
LeakyRelu backward incorrect. Expected: < 1e-8 Evaluated: 1.0
Test cases are still failing! Tests passed: 1/2
Score: 50/100
You secured a score of :50
```

And this cell to test your implementation of the `Tanh` class:

In [12]:
```
print(TanhTestWrapper()())
```

```
TanhForwardTest passed.
TanhBackwardTest passed.
Congratulations you have passed all the unit tests!!! Tests passed: 2/2
Score: 100/100
You secured a score of :100
```

Congratulations, you now implemented all four different activation functions! These activation layers are now ready to be used when you start building your own network.

1.4 Loss

In order to measure how well a network is performing, we implemented several `Loss` classes (`L1`, `MSE`, `BCE`, each preferred for a certain type of problems) in `exercise_code/networks/loss.py`.

Each implemented a `forward()` method, which outputs a number that we use as a proxy for our network performance.

Also, because our goal was to change the weights of the network such that this loss measure decreases, we were also interested in the gradients of the loss w.r.t the outputs of the network, $\nabla_{\hat{y}} L(\hat{y}, y)$. This was implemented in `backward()`.

In previous exercises, we only worked with binary classification and used binary cross entropy (`BCE`) as a loss function.

$$BCE(\hat{y}, y) = \frac{1}{N} \sum_{i=1}^{N} \Big[- y_i \log(\hat{y_i}) - (1 - y_i) \log(1 - \hat{y_i}) \Big]$$

where

- N was the number of samples we were considering
- $\hat{y}_i$ was the network's prediction for sample i. Note that this was a valid probability $\in [0, 1]$, because we applied a sigmoid (https://en.wikipedia.org/wiki/Sigmoid_function) activation on the last layer.
- y_i was the ground truth label (0 or 1, depending on the class)

Because we have 10 classes in the CIFAR10 dataset, we need a generalization of the binary cross entropy for multiple classes. This is simply called the cross entropy loss and has the following definition:

$$CE(\hat{y}, y) = \frac{1}{N} \sum_{i=1}^{N} \sum_{k=1}^{C} \Big[- y_{ik} \log(\hat{y}_{ik}) \Big]$$

where:

- N is again the number of samples
- C is the number of classes
- $\hat{y}_{ik}$ is the probability that the model assigns for the kth class when the ith sample is the input. **Because we don't apply any activation function on the last layer of our network, its outputs for each sample will not be a valid probability distribution over the classes. We call these raw outputs of the network 'logits (https://datascience.stackexchange.com/questions/31041/what-does-logits-in-machine-learning-mean/31045)' and we will apply a softmax (https://en.wikipedia.org/wiki/Softmax_function) activation in order to obtain a valid probability distribution.**
- $y_{ik} = 1$ iff the true label of the ith sample is k and 0 otherwise. This is called a one-hot encoding (https://machinelearningmastery.com/why-one-hot-encode-data-in-machine-learning/).

You can check for yourself that when the number of classes C is 2, then the binary cross entropy is actually equivalent to the cross entropy.

Task: Check Code

Please check the implementation of the `CrossEntropyFromLogits` class, which can be found in `../exercise_06/exercise_code/networks/loss.py`.

In [36]:
```
loss = CrossEntropyFromLogits()
```

Just as with the previous losses we have seen, we can simply get the results of the forward and backward passes as follows:

```
# y_out is the output of the neural network
# y_truth is the actual label from the dataset
loss.forward(y_out, y_truth)
loss.backward(y_out, y_truth)
```

1.5 Optimizer

Now, knowing the gradient of the loss w.r.t the ouputs of the network, as well as the local gradient for each layer of the network, we can use the chain rule to compute all gradients.

Task: Check Code

We implemented several optimizer classes `SGD`, `Adam`, `sgd_momentum` class that implement different first-order parameter update rules, which can be found in `../exercise_06/exercise_code/networks/optimizer.py`.

The `step()` method iterates through all the parameters of a model and updates them using the gradient information.

What the optimizer is doing, in pseudocode, is the following:

```
for param in model:
    # Use the gradient to update the weights.
    update(param)

    # Reset the gradient after each update.
    param.gradient = 0
```

`SGD` had the simplest update rule:

```
def update(param):
    param = param - learning_rate * param.gradient
```

For the more complicated update rules, see `exercise_code/networks/optimizer.py`

1.6 Solver

The `Solver` is where all the above elements come together: Given a train and a validation dataloader, a model, a loss and an optimizer, it uses the training data to optimize a model in order to get better predictions. We simply call `train()` and it does its 'magic' for us!

```
solver = Solver(model,
                dataloaders['train'],
                dataloaders['val'],
                learning_rate=0.001,
                loss_func=MSE(),
                optimizer=SGD)

solver.train(epochs=epochs)
```

Task: Check Code

Please check out the implementation of `train()` in `../exercise_06/exercise_code/solver.py`.

1.7 Weight Regularization

Before we finish this section of recap, we want to take a look at some regularization method that has been introduced in the lecture and that is super helpful to improve robustness of our model. Here, we talk about weight regularization.

Weight regularization has been introduced to you as a method preventing our model from overfitting. Essentially, it is a term (solely depending on the weights of our model) that is added to the final loss and that encodes some preference for a certain set of weights W over others. In the lecture, we compared two weight regularization methods and their respective preference for weight vectors. We made the following observation:

1. L1 regularization: Enforces sparsity
2. L2 regularization: Enforces that weights have similar values

The most common weight regularization method is the L2 regularization. From the observations made in the lecture that makes totally sense - at least when we compare it to the L1 regularization. The L2 regularization penalty in the loss prefers smaller and more diffuse weight vectors and hence the model is encouraged to take into account all input dimensions to small amounts rather than a few input dimensions and very strongly.

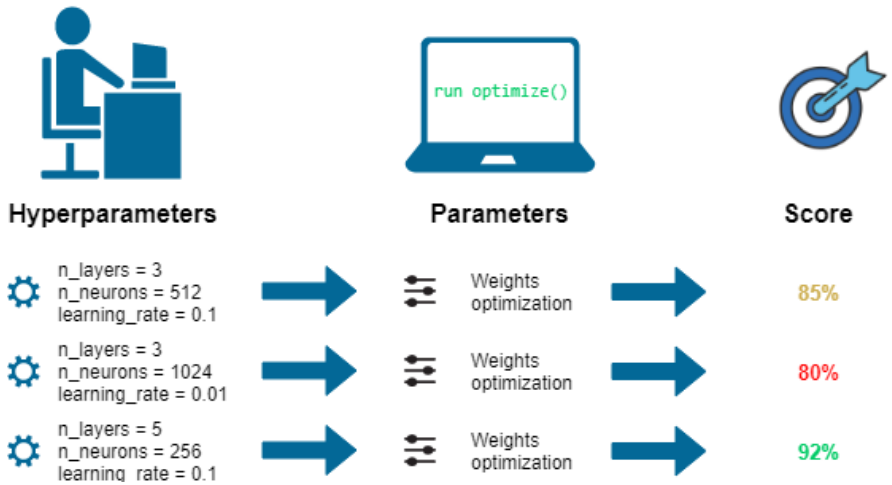
When using weight regularization, the loss function is a composition of two parts:

$$L = \underbrace{\frac{1}{N}\sum_i L_i}_{\text{data loss}} + \underbrace{\lambda R(W)}_{\text{regularization loss}}$$

The first one being the data loss, which is calculated with the Cross Entropy loss in our model. The second part is called the regularization loss $R(W)$ and is computed in the L2 case as follows:

$$R(W) = \sum_k \sum_l w_{k,l}^2$$

2. An overview of hyperparameters



A **hyperparameter** is a parameter that is set before the learning process begins. Recall that the parameters of weight matrix and bias vector are learned during the learning process.

The hyperparameters are essential, for they control and affect the whole training and have a great impact on the performance of the model.

Some examples of hyperparameters we have covered in lectures:

- Network architecture
  - Choice of activation function
  - Number of layers
  - ...
- Learning rate
- Number of epochs
- Batch size
- Regularization strength
- Momentum
- ...

2.1 Start debugging your own network!

As already suggested in the lectures, you may always want to start from small and simple architectures, to make sure you are going the right way.

we can edit our own network

First you may need to overfit a single training sample, then a few batches of training samples, then go deeper with larger neural networks and the whole training data.

Here we always provide a default neural network (i.e. ClassificationNet) with arbitrary number of layers, which is a generalization from a fixed 2-layer neural network in exercise 5. You are welcome to implement your own network, in that case just implement **MyOwnNetwork** in `exercise_code/networks/classification_net.py` . You can also copy things from ClassficationNet and make a little adjustment to your own network. For either way, just pick a network and comment out the other one, then run the cells below for debugging.

> **Note:**
>
> Please, make sure you don't modify the ClassificationNet itself so that you can always have a working network to fall back on
>
> In order to pass this submissions, you can **first stick to the default ClassificationNet implementation without changing any code at all**. The goal of this submission is to find reasonable hyperparameters and the parameter options of the ClassificationNet are broad enough.
>
> Once you have surpassed the submission goal, you can try to implement additional activation functions in the accompanying notebook, try different weight initializations or other adjustments by writing your own network architecture in the MyOwnNetwork class.

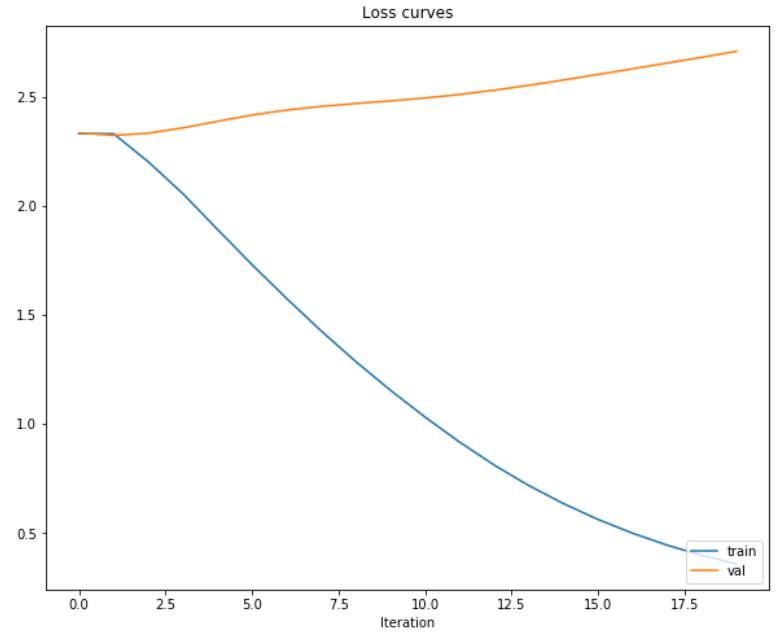
First, let's begin with a 2-layer neural network, and overfit one single training sample.

After training, let's evaluate the training process by plotting the loss curves.

In [37]:
```
from exercise_code.solver import Solver
from exercise_code.networks.optimizer import SGD, Adam
from exercise_code.networks import MyOwnNetwork

num_layer = 2
epochs = 20
reg = 0.1
batch_size = 4

model = ClassificationNet(num_layer=num_layer, reg=reg)
# model = MyOwnNetwork()

loss = CrossEntropyFromLogits()

# Make a new data loader with a single training image
overfit_dataset = ImageFolderDataset(
    mode='train',
    root=cifar_root,
    download_url=download_url,
    transform=compose_transform,
    limit_files=1
)
dataloaders['train_overfit_single_image'] = DataLoader(
    dataset=overfit_dataset,
    batch_size=batch_size,
    shuffle=True,
    drop_last=False,
)

# Decrease validation data for only debugging
debugging_validation_dataset = ImageFolderDataset(
    mode='val',
    root=cifar_root,
    download_url=download_url,
    transform=compose_transform,
    limit_files=100
)
dataloaders['val_500files'] = DataLoader(
    dataset=debugging_validation_dataset,
    batch_size=batch_size,
    shuffle=True,
    drop_last=True,
)

solver = Solver(model, dataloaders['train_overfit_single_image'], datalo
aders['val_500files'],
                learning_rate=1e-3, loss_func=loss, optimizer=Adam)

solver.train(epochs=epochs)
```

```
(Epoch 1 / 20) train loss: 2.329775; val loss: 2.333335
(Epoch 2 / 20) train loss: 2.329775; val loss: 2.322043
(Epoch 3 / 20) train loss: 2.201227; val loss: 2.331887
(Epoch 4 / 20) train loss: 2.055510; val loss: 2.356117
(Epoch 5 / 20) train loss: 1.891945; val loss: 2.386362
(Epoch 6 / 20) train loss: 1.729935; val loss: 2.414995
(Epoch 7 / 20) train loss: 1.575250; val loss: 2.437885
(Epoch 8 / 20) train loss: 1.426365; val loss: 2.454867
(Epoch 9 / 20) train loss: 1.285956; val loss: 2.467926
(Epoch 10 / 20) train loss: 1.154995; val loss: 2.479860
(Epoch 11 / 20) train loss: 1.032151; val loss: 2.493155
(Epoch 12 / 20) train loss: 0.917189; val loss: 2.509355
(Epoch 13 / 20) train loss: 0.812081; val loss: 2.528878
(Epoch 14 / 20) train loss: 0.718319; val loss: 2.551226
(Epoch 15 / 20) train loss: 0.635641; val loss: 2.575500
(Epoch 16 / 20) train loss: 0.563106; val loss: 2.600866
(Epoch 17 / 20) train loss: 0.499872; val loss: 2.626770
(Epoch 18 / 20) train loss: 0.445184; val loss: 2.652980
(Epoch 19 / 20) train loss: 0.398510; val loss: 2.679587
(Epoch 20 / 20) train loss: 0.359475; val loss: 2.706924
```


In [38]:
```
plt.title('Loss curves')
plt.plot(solver.train_loss_history, '-', label='train')
plt.plot(solver.val_loss_history, '-', label='val')
plt.legend(loc='lower right')
plt.xlabel('Iteration')
plt.show()
```



In [39]:
```
print("Training accuray: %.5f" % (solver.get_dataset_accuracy(dataloader
s['train_overfit_single_image'])))
print("Validation accuray: %.5f" % (solver.get_dataset_accuracy(dataload
ers['val_500files'])))
```

```
Training accuray: 1.00000
Validation accuray: 0.07000
```

This time let's try to overfit to a small set of training batch samples. Please observe the difference from above.

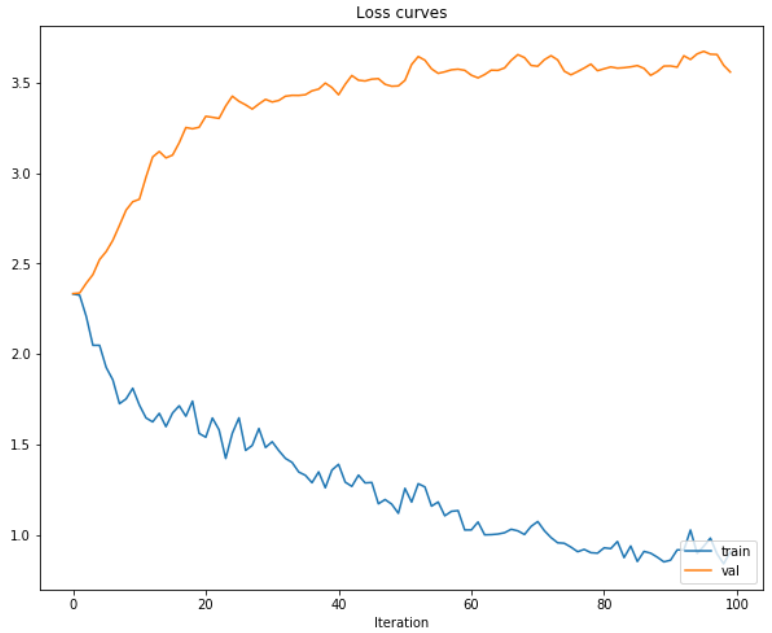
In [40]:
```
from exercise_code.networks import MyOwnNetwork

num_layer = 2
epochs = 100
reg = 0.1
num_samples = 10

model = ClassificationNet(num_layer=num_layer, reg=reg)
# model = MyOwnNetwork()

loss = CrossEntropyFromLogits()

# Make a new data loader with a our num_samples training image
overfit_dataset = ImageFolderDataset(
    mode='train',
    root=cifar_root,
    download_url=download_url,
    transform=compose_transform,
    limit_files=num_samples
)
dataloaders['train_overfit_10samples'] = DataLoader(
    dataset=overfit_dataset,
    batch_size=batch_size,
    shuffle=True,
    drop_last=False,
)

solver = Solver(model, dataloaders['train_overfit_10samples'], dataloade
rs['val_500files'],
                learning_rate=1e-3, loss_func=loss, optimizer=Adam)

solver.train(epochs=epochs)
```



```
(Epoch 1 / 100) train loss: 2.331938; val loss: 2.334331
(Epoch 2 / 100) train loss: 2.325609; val loss: 2.337541
(Epoch 3 / 100) train loss: 2.207519; val loss: 2.391639
(Epoch 4 / 100) train loss: 2.048381; val loss: 2.439445
(Epoch 5 / 100) train loss: 2.048479; val loss: 2.522273
(Epoch 6 / 100) train loss: 1.925998; val loss: 2.566617
(Epoch 7 / 100) train loss: 1.857588; val loss: 2.628791
(Epoch 8 / 100) train loss: 1.725757; val loss: 2.711391
(Epoch 9 / 100) train loss: 1.752357; val loss: 2.795958
(Epoch 10 / 100) train loss: 1.810925; val loss: 2.842225
(Epoch 11 / 100) train loss: 1.718056; val loss: 2.855684
(Epoch 12 / 100) train loss: 1.646908; val loss: 2.979115
(Epoch 13 / 100) train loss: 1.625130; val loss: 3.089495
(Epoch 14 / 100) train loss: 1.672408; val loss: 3.119818
(Epoch 15 / 100) train loss: 1.598648; val loss: 3.084452
(Epoch 16 / 100) train loss: 1.674407; val loss: 3.100027
(Epoch 17 / 100) train loss: 1.714083; val loss: 3.168348
(Epoch 18 / 100) train loss: 1.656148; val loss: 3.252408
(Epoch 19 / 100) train loss: 1.739993; val loss: 3.245475
(Epoch 20 / 100) train loss: 1.561257; val loss: 3.253790
(Epoch 21 / 100) train loss: 1.540376; val loss: 3.314405
(Epoch 22 / 100) train loss: 1.646614; val loss: 3.308773
(Epoch 23 / 100) train loss: 1.581151; val loss: 3.302904
(Epoch 24 / 100) train loss: 1.422976; val loss: 3.371343
(Epoch 25 / 100) train loss: 1.562293; val loss: 3.425782
(Epoch 26 / 100) train loss: 1.647568; val loss: 3.397219
(Epoch 27 / 100) train loss: 1.467402; val loss: 3.377734
(Epoch 28 / 100) train loss: 1.495070; val loss: 3.354369
(Epoch 29 / 100) train loss: 1.588650; val loss: 3.382466
(Epoch 30 / 100) train loss: 1.483492; val loss: 3.407889
(Epoch 31 / 100) train loss: 1.515264; val loss: 3.393643
(Epoch 32 / 100) train loss: 1.466285; val loss: 3.401993
(Epoch 33 / 100) train loss: 1.423159; val loss: 3.425434
(Epoch 34 / 100) train loss: 1.400972; val loss: 3.430258
(Epoch 35 / 100) train loss: 1.348053; val loss: 3.429814
(Epoch 36 / 100) train loss: 1.329759; val loss: 3.433689
(Epoch 37 / 100) train loss: 1.288834; val loss: 3.455196
(Epoch 38 / 100) train loss: 1.348734; val loss: 3.464836
(Epoch 39 / 100) train loss: 1.260980; val loss: 3.498036
(Epoch 40 / 100) train loss: 1.358767; val loss: 3.473409
(Epoch 41 / 100) train loss: 1.390493; val loss: 3.433425
(Epoch 42 / 100) train loss: 1.292045; val loss: 3.491149
(Epoch 43 / 100) train loss: 1.268652; val loss: 3.539978
(Epoch 44 / 100) train loss: 1.330824; val loss: 3.513853
(Epoch 45 / 100) train loss: 1.287729; val loss: 3.509725
(Epoch 46 / 100) train loss: 1.290267; val loss: 3.519821
(Epoch 47 / 100) train loss: 1.172291; val loss: 3.522474
(Epoch 48 / 100) train loss: 1.195694; val loss: 3.491398
(Epoch 49 / 100) train loss: 1.170439; val loss: 3.480601
(Epoch 50 / 100) train loss: 1.119711; val loss: 3.482389
(Epoch 51 / 100) train loss: 1.257560; val loss: 3.513039
(Epoch 52 / 100) train loss: 1.181583; val loss: 3.600747
(Epoch 53 / 100) train loss: 1.283340; val loss: 3.644795
(Epoch 54 / 100) train loss: 1.265615; val loss: 3.624526
(Epoch 55 / 100) train loss: 1.159158; val loss: 3.576834
(Epoch 56 / 100) train loss: 1.182414; val loss: 3.551872
(Epoch 57 / 100) train loss: 1.105850; val loss: 3.560429
```

```
(Epoch 58 / 100) train loss: 1.130616; val loss: 3.571432
(Epoch 59 / 100) train loss: 1.134558; val loss: 3.574873
(Epoch 60 / 100) train loss: 1.027422; val loss: 3.568672
(Epoch 61 / 100) train loss: 1.027777; val loss: 3.541614
(Epoch 62 / 100) train loss: 1.071632; val loss: 3.526807
(Epoch 63 / 100) train loss: 1.000724; val loss: 3.545189
(Epoch 64 / 100) train loss: 1.001871; val loss: 3.569452
(Epoch 65 / 100) train loss: 1.005035; val loss: 3.568322
(Epoch 66 / 100) train loss: 1.012345; val loss: 3.581639
(Epoch 67 / 100) train loss: 1.031481; val loss: 3.623874
(Epoch 68 / 100) train loss: 1.022743; val loss: 3.655385
(Epoch 69 / 100) train loss: 1.002597; val loss: 3.638895
(Epoch 70 / 100) train loss: 1.047481; val loss: 3.595655
(Epoch 71 / 100) train loss: 1.074237; val loss: 3.591181
(Epoch 72 / 100) train loss: 1.023344; val loss: 3.627247
(Epoch 73 / 100) train loss: 0.985536; val loss: 3.649597
(Epoch 74 / 100) train loss: 0.956673; val loss: 3.626058
(Epoch 75 / 100) train loss: 0.953968; val loss: 3.563884
(Epoch 76 / 100) train loss: 0.932819; val loss: 3.543700
(Epoch 77 / 100) train loss: 0.907463; val loss: 3.561685
(Epoch 78 / 100) train loss: 0.920114; val loss: 3.581150
(Epoch 79 / 100) train loss: 0.902032; val loss: 3.603120
(Epoch 80 / 100) train loss: 0.899427; val loss: 3.566424
(Epoch 81 / 100) train loss: 0.928899; val loss: 3.577112
(Epoch 82 / 100) train loss: 0.924480; val loss: 3.587221
(Epoch 83 / 100) train loss: 0.964044; val loss: 3.580696
(Epoch 84 / 100) train loss: 0.874808; val loss: 3.583204
(Epoch 85 / 100) train loss: 0.938830; val loss: 3.587788
(Epoch 86 / 100) train loss: 0.853180; val loss: 3.594803
(Epoch 87 / 100) train loss: 0.909091; val loss: 3.579225
(Epoch 88 / 100) train loss: 0.899377; val loss: 3.540678
(Epoch 89 / 100) train loss: 0.877273; val loss: 3.561657
(Epoch 90 / 100) train loss: 0.851209; val loss: 3.592029
(Epoch 91 / 100) train loss: 0.860576; val loss: 3.592714
(Epoch 92 / 100) train loss: 0.917568; val loss: 3.586197
(Epoch 93 / 100) train loss: 0.916384; val loss: 3.648954
(Epoch 94 / 100) train loss: 1.027391; val loss: 3.628603
(Epoch 95 / 100) train loss: 0.898212; val loss: 3.659187
(Epoch 96 / 100) train loss: 0.937496; val loss: 3.673554
(Epoch 97 / 100) train loss: 0.982593; val loss: 3.657673
(Epoch 98 / 100) train loss: 0.895751; val loss: 3.655952
(Epoch 99 / 100) train loss: 0.840275; val loss: 3.596174
(Epoch 100 / 100) train loss: 0.906998; val loss: 3.558889
```


In [41]:
```
plt.title('Loss curves')
plt.plot(solver.train_loss_history, '-', label='train')
plt.plot(solver.val_loss_history, '-', label='val')
plt.legend(loc='lower right')
plt.xlabel('Iteration')
plt.show()
```



In [42]:
```
print("Training accuray: %.5f" % (solver.get_dataset_accuracy(dataloader
s['train_overfit_10samples'])))
print("Validation accuray: %.5f" % (solver.get_dataset_accuracy(dataload
ers['val_500files'])))
```

```
Training accuray: 1.00000
Validation accuray: 0.06000
```

If you're overfitting the training data, that means the network's implementation is correct. However, as you have more samples to overfit, your accuracy will be way lower. You can increase the number of epochs above to achieve better results.

Now let's try to feed all the training and validation data into the network, but this time we set the same hyperparameters for 2-layer and 5-layer networks, and compare the different behaviors.

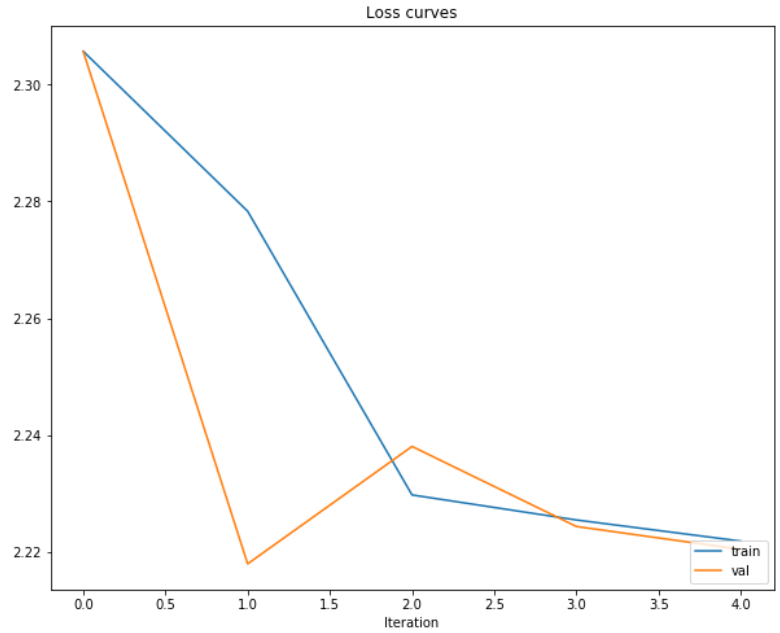
Note:

This may take about 1 min for each epoch as the training set is quite large. For convenience, we only train on 1000 images for now but use the full validation set.

In [43]:
```
from exercise_code.networks import MyOwnNetwork

num_layer = 2
epochs = 5
reg = 0.01

# Make a new data loader with 1000 training samples
num_samples = 1000
overfit_dataset = ImageFolderDataset(
    mode='train',
    root=cifar_root,
    download_url=download_url,
    transform=compose_transform,
    limit_files=num_samples
)
dataloaders['train_small'] = DataLoader(
    dataset=overfit_dataset,
    batch_size=batch_size,
    shuffle=True,
    drop_last=False,
)


# Change here if you want to use the full training set
use_full_training_set = False
if not use_full_training_set:
    train_loader = dataloaders['train_small']
else:
    train_loader = dataloaders['train']


model = ClassificationNet(num_layer=num_layer, reg=reg)
# model = MyOwnNetwork()

loss = CrossEntropyFromLogits()

solver = Solver(model, train_loader, dataloaders['val'],
                learning_rate=1e-3, loss_func=loss, optimizer=Adam)

solver.train(epochs=epochs)
```

```
(Epoch 1 / 5) train loss: 2.305701; val loss: 2.305666
(Epoch 2 / 5) train loss: 2.278313; val loss: 2.217917
(Epoch 3 / 5) train loss: 2.229720; val loss: 2.238025
(Epoch 4 / 5) train loss: 2.225433; val loss: 2.224318
(Epoch 5 / 5) train loss: 2.221815; val loss: 2.220396
```

In [44]:
```
plt.title('Loss curves')
plt.plot(solver.train_loss_history, '-', label='train')
plt.plot(solver.val_loss_history, '-', label='val')
plt.legend(loc='lower right')
plt.xlabel('Iteration')
plt.show()
```



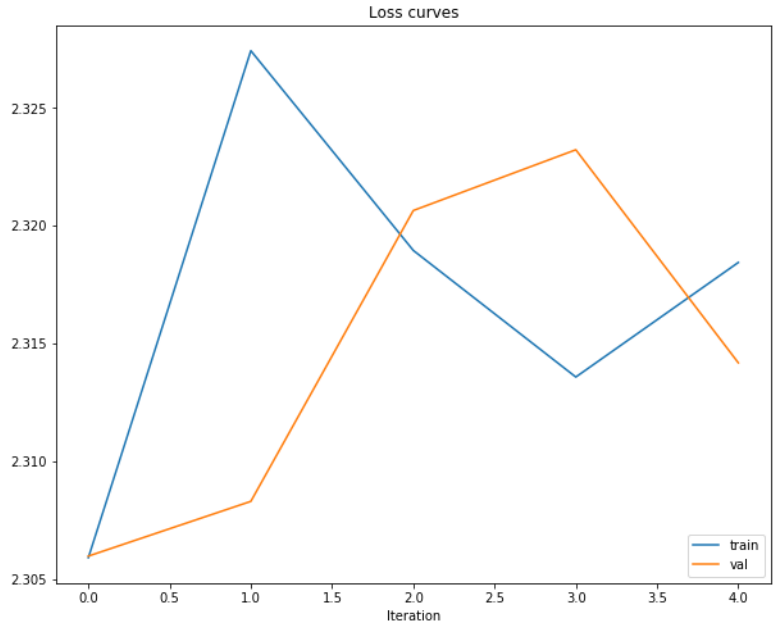
In [45]:
```
print("Training accuray: %.5f" % (solver.get_dataset_accuracy(train_load
er)))
print("Validation accuray: %.5f" % (solver.get_dataset_accuracy(dataload
ers['val'])))
```

```
Training accuray: 0.31800
Validation accuray: 0.26002
```


In [46]:
```
from exercise_code.networks import MyOwnNetwork

num_layer = 5
epochs = 5
reg = 0.01

model = ClassificationNet(num_layer=num_layer, reg=reg)
# model = MyOwnNetwork()

# Change here if you want to use the full training set
use_full_training_set = False
if not use_full_training_set:
    train_loader = dataloaders['train_small']
else:
    train_loader = dataloaders['train']

loss = CrossEntropyFromLogits()

solver = Solver(model, train_loader, dataloaders['val'],
                learning_rate=1e-3, loss_func=loss, optimizer=Adam)

solver.train(epochs=epochs)
```

```
(Epoch 1 / 5) train loss: 2.305912; val loss: 2.305979
(Epoch 2 / 5) train loss: 2.327428; val loss: 2.308302
(Epoch 3 / 5) train loss: 2.318951; val loss: 2.320646
(Epoch 4 / 5) train loss: 2.313579; val loss: 2.323224
(Epoch 5 / 5) train loss: 2.318441; val loss: 2.314183
```

In [47]:
```
plt.title('Loss curves')
plt.plot(solver.train_loss_history, '-', label='train')
plt.plot(solver.val_loss_history, '-', label='val')
plt.legend(loc='lower right')
plt.xlabel('Iteration')
plt.show()
```



In [48]:
```
print("Training accuray: %.5f" % (solver.get_dataset_accuracy(train_load
er)))
print("Validation accuray: %.5f" % (solver.get_dataset_accuracy(dataload
ers['val'])))
```

```
Training accuray: 0.10500
Validation accuray: 0.09826
```

As you can see from above, the same hyperparameter set can decrease the loss for a 2-layer network, but for 5-layer network, it hardly works.

The steps above are already mentioned in the lectures as debugging steps before training a neural network.

If you implement your own network, make sure you do the steps above before tuning the hyperparameters as below.

2.2 Difficulty in tuning hyperparameters

As can be seen through the results of training a larger network, training with whole data doesn't fit the training data as well as training with small number of training data. Besides, the architecture of neural network makes a difference, too. Small decisions on hyperparameters count.

Usually, but not always, hyperparameters cannot be learned using well known gradient based methods (such as gradient descent), which are commonly employed to learn parameters. Besides, some hyperparameters can affect the structure of the model and the loss function.

As mentioned before, hyperparameters need to be set before training. Tuning hyperparameters is hard, because you always have to try different combinations of the hyperparameters, train the network, do the validation and pick the best one. Besides, it is not guaranteed that you'll find the best combination.

Next you will do hands on learning with hyperparameter tuning methods that are covered in lectures.

3. Hyperparameter Tuning

One of the main challenges in deep learning is finding the set of hyperparameters that performs best.

So far, we have followed a manual approach by guessing hyperparameters, running the model, observing the result and maybe tweaking the hyperparameters based on this result. As you have probably noticed, this manual hyperparameter tuning is unstructured, inefficient and can become very tedious.

A more systematic (and actually very simple) approach for hyperparameter tuning that you've already learned in the lecture is implementing a **Grid Search**.

3.1 Grid Search

Grid search is a simple and naive, yet effective method to automate the hyperparameter tuning:

- First, you define the set of parameters you want to tune, e.g. $\{learning\_rate, regularization\_strength\}$.
- For each hyperparameter, you then define a set of possible values, e.g. $learning\_rate = \{0.0001, 0.001, 0.01, 0.1\}$.
- Then, you train a model for every possible combination of these hyperparameter values and afterwards select the combination that works best (e.g. in terms of accuracy on your validation set).

Task: Check Code

Check out our `grid_search` implementation in `../exercise_6/exercise_code/hyperparameter_tuning.py`. We show a simple for loop implementation and a more sophisticated one for multiple inputs.

Note:

To keep things simple for the beginning, it'll be enough to just focus on the hyperparameters `learning_rate` and `regularization_strength` here, as in the example above.

In [49]:
```
from exercise_code.networks import MyOwnNetwork

# Specify the used network
model_class = ClassificationNet

from exercise_code import hyperparameter_tuning
best_model, results = hyperparameter_tuning.grid_search(
    dataloaders['train_small'], dataloaders['val_500files'],
    grid_search_spaces = {
        "learning_rate": [1e-2, 1e-3, 1e-4],
        "reg": [1e-4]
    },
    model_class=model_class,
    epochs=10, patience=5)
```

```
Evaluating Config #1 [of 3]:
 {'learning_rate': 0.01, 'reg': 0.0001}
(Epoch 1 / 10) train loss: 2.302778; val loss: 2.302379
(Epoch 2 / 10) train loss: 2.552255; val loss: 2.538771
(Epoch 3 / 10) train loss: 2.745131; val loss: 2.816231

/Users/meiqiliu/Downloads/exercise_06/exercise_code/networks/layer.py:6
7: RuntimeWarning: overflow encountered in exp
  outputs = 1 / (1 + np.exp(-x))

(Epoch 4 / 10) train loss: 2.920796; val loss: 3.092683
(Epoch 5 / 10) train loss: 3.041066; val loss: 3.151139
(Epoch 6 / 10) train loss: 3.174544; val loss: 3.260573
Stopping early at epoch 5!

Evaluating Config #2 [of 3]:
 {'learning_rate': 0.001, 'reg': 0.0001}
(Epoch 1 / 10) train loss: 2.302776; val loss: 2.302850
(Epoch 2 / 10) train loss: 2.188627; val loss: 2.018576
(Epoch 3 / 10) train loss: 1.999532; val loss: 1.933551
(Epoch 4 / 10) train loss: 1.898786; val loss: 1.913170
(Epoch 5 / 10) train loss: 1.842902; val loss: 1.897408
(Epoch 6 / 10) train loss: 1.767757; val loss: 1.882606
(Epoch 7 / 10) train loss: 1.710118; val loss: 1.961722
(Epoch 8 / 10) train loss: 1.636581; val loss: 1.945361
(Epoch 9 / 10) train loss: 1.585429; val loss: 1.916295
(Epoch 10 / 10) train loss: 1.516268; val loss: 1.924506

Evaluating Config #3 [of 3]:
 {'learning_rate': 0.0001, 'reg': 0.0001}
(Epoch 1 / 10) train loss: 2.302742; val loss: 2.302700
(Epoch 2 / 10) train loss: 2.286272; val loss: 2.243366
(Epoch 3 / 10) train loss: 2.212058; val loss: 2.153702
(Epoch 4 / 10) train loss: 2.136847; val loss: 2.086200
(Epoch 5 / 10) train loss: 2.075843; val loss: 2.034195
(Epoch 6 / 10) train loss: 2.025140; val loss: 1.996724
(Epoch 7 / 10) train loss: 1.980983; val loss: 1.980872
(Epoch 8 / 10) train loss: 1.941590; val loss: 1.947735
(Epoch 9 / 10) train loss: 1.904624; val loss: 1.928624
(Epoch 10 / 10) train loss: 1.870713; val loss: 1.909161

Search done. Best Val Loss = 1.8826062344214936
Best Config: {'learning_rate': 0.001, 'reg': 0.0001}
```


From the results of your grid search, you might already have found some hyperparameter combinations that work better than others. A common practice is to now repeat the grid search on a more narrow domain centered around the parameters that worked best.

Conclusion Grid Search

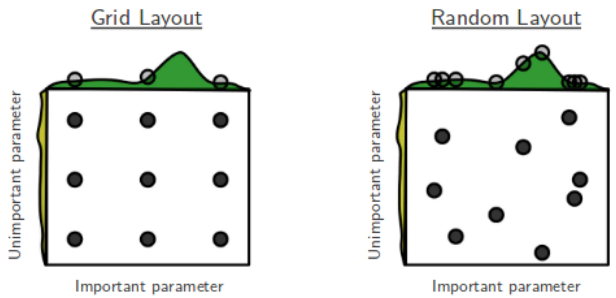
With grid search we now have automated the hyperparameter tuning to a certain degree. Another advantage is, that since the training of all models are independent of each other, you can parallelize the grid search, i.e., try out different hyperparameter configurations in parallel on different machines.

However, as you have probably noticed, there is one big problem with this approach: the number of possible combinations to try out grows exponentially with the number of hyperparameters ("curse of dimensionality"). As we add more hyperparameters to the grid search, the search space will explode in time complexity, making this strategy unfeasible.

Especially when your search space contains more than 3 or 4 dimensions, it is often better to use another, similar hyperparameter tuning method that you've already learned about: random search.

3.2 Random Search

Random search is very similar to grid search, with the only difference, that instead of providing specific values for every hyperparameter, you only define a range for each hyperparameter - then, the values are sampled randomly from the provided ranges.



The figure above illustrates the difference in the hyperparameter space exploration between grid search and random search: assume you have 2 hyperparameters with each 3 values. Running a grid search results in training $3^2 = 9$ different models - but in the end, you've just tired out 3 values for each parameter. For random search on the other hand, after training 9 models you'll have tried out 9 different values for each hyperparameter, which often leads much faster to good results.

To get a deeper understanding of random search and why it is more efficient than grid search, you should definitely check out this paper: http://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.pdf (http://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.pdf)

Task: Check Code

Check out our `random_search` implementation in
`../exercise_6/exercise_code/hyperparameter_tuning.py`

Hint: regarding the sample space of each parameter, think about the scale for which it makes most sense to sample in. For example the learning rate is usually sampled on a logarithmic scale!

For simplicity and speed, just use the `train_batches` -dataloader!

In [50]:
```
from exercise_code.hyperparameter_tuning import random_search
from exercise_code.networks import MyOwnNetwork

# Specify the used network
model_class = ClassificationNet

best_model, results = random_search(
    dataloaders['train_small'], dataloaders['val_500files'],
    random_search_spaces = {
        "learning_rate": ([1e-2, 1e-6], 'log'),
        "reg": ([1e-3, 1e-7], "log"),
        "loss_func": ([CrossEntropyFromLogits()], "item")
    },
    model_class=model_class,
    num_search = 1, epochs=20, patience=5)
```

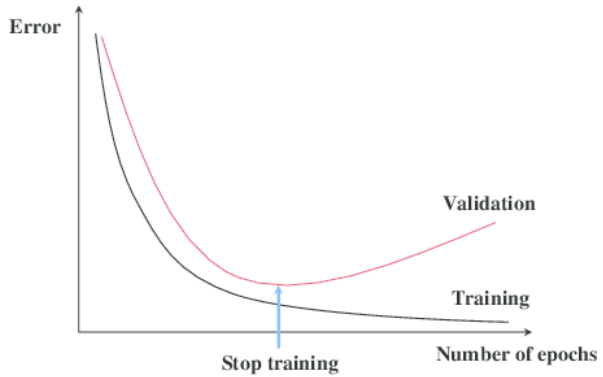
```
Evaluating Config #1 [of 1]:
 {'learning_rate': 1.1180385218225658e-05, 'reg': 0.0001351741379270533
3, 'loss_func': <exercise_code.networks.loss.CrossEntropyFromLogits obj
ect at 0x7fe159a4e2d0>}
(Epoch 1 / 20) train loss: 2.302957; val loss: 2.302439
(Epoch 2 / 20) train loss: 2.302167; val loss: 2.300204
(Epoch 3 / 20) train loss: 2.299066; val loss: 2.296942
(Epoch 4 / 20) train loss: 2.294483; val loss: 2.291335
(Epoch 5 / 20) train loss: 2.288183; val loss: 2.283739
(Epoch 6 / 20) train loss: 2.280328; val loss: 2.274315
(Epoch 7 / 20) train loss: 2.271355; val loss: 2.263522
(Epoch 8 / 20) train loss: 2.261663; val loss: 2.252223
(Epoch 9 / 20) train loss: 2.251528; val loss: 2.240728
(Epoch 10 / 20) train loss: 2.241236; val loss: 2.228350
(Epoch 11 / 20) train loss: 2.230858; val loss: 2.217009
(Epoch 12 / 20) train loss: 2.220711; val loss: 2.205581
(Epoch 13 / 20) train loss: 2.210612; val loss: 2.195147
(Epoch 14 / 20) train loss: 2.200769; val loss: 2.184289
(Epoch 15 / 20) train loss: 2.190997; val loss: 2.174719
(Epoch 16 / 20) train loss: 2.181389; val loss: 2.164967
(Epoch 17 / 20) train loss: 2.172101; val loss: 2.155849
(Epoch 18 / 20) train loss: 2.163095; val loss: 2.147377
(Epoch 19 / 20) train loss: 2.154309; val loss: 2.139036
(Epoch 20 / 20) train loss: 2.145777; val loss: 2.131252

Search done. Best Val Loss = 2.131252055117088
Best Config: {'learning_rate': 1.1180385218225658e-05, 'reg': 0.0001351
7413792705333, 'loss_func': <exercise_code.networks.loss.CrossEntropyFr
omLogits object at 0x7fe159a4e2d0>}
```

It's time to run it with the whole dataset, and let it search for a few hours for a nice configuration.

However, to save some time, let's first implement an **early-stopping** mechanism, that you also already know from the lecture.

3.3 Early Stopping

By now you've already seen a lot of training curves:



Usually, at some point the validation loss goes up again, which is a sign that we're overfitting to our training data. Since it actually doesn't make any sense to train further at this point, it's common practice to apply "early stopping", i.e., cancel the training process when the validation loss doesn't improve anymore. The nice thing about this concept is, that not only it improves generalization through the prevention of overfitting, but also it saves us a lot of time - one of our most valuable resources in deep learning.

Since there are natural fluctuations in the validation loss, you usually don't cancel the training process right at the first epoch when the validation-loss increases, but instead, you wait for some epochs (specified by the `patience` -parameter) and if the loss still doesn't improve, we stop.

Task: Check Code

Please check the implementation of the early stopping mechanism in `../exercise_6/exercise_code/solver.py` .

3.4 Let's find the perfect model!

You've now set everything up to start training your model and finding a nice set of hyper parameters using a combination of grid or random search!

Since we'll now be training with a much larger number of samples, you should be aware that this process will definitely take some time! So be prepared to let your machine run for a while.

At the beginning, it's a good approach to first do a coarse random search across a wide range of values to find promising sub-ranges of your parameter space. Afterwards, you can zoom in to these ranges and do another random search (or grid search) to finetune the configuration.

You don't have to use the whole dataset at the beginning, instead you can also use a medium large subset of the samples. Also, you don't need to train for a large number of epochs - as mentioned above: we first want to get an overview about our hyper parameters.

Task: Hyperparameters Tunning & Model Training

Now, it is your turn to do the hyperparamater tuning. In the cell below, you can use the `random_search` function to find a good choice of parameters. Put in some reasonable ranges for the hyperparameters and evaluate them.

Note: At the beginning, it's a good approach to first do a coarse random search across a **wide range of values** to find promising sub-ranges of your parameter space and use **a medium large subset of the dataset** instead the whole as well. Afterwards, you can zoom in to these ranges and do another random search (or grid search) to finetune the configuration. Use the cell below to play around and find good hyperparameters for your model!

Finally, once you've found some promising hyperparameters (or narrowed them down to promising subranges), it's time to utilize these hyperparameters to train your network on the whole dataset for a large number of epochs so that your own model can reach an acceptable performance.

Hint: You may use a `Solver` class we provided before or directly use the `random_search` function (as you can also monitor the loss here) for model training.

In [51]:
```
from exercise_code.networks import MyOwnNetwork

best_model = ClassificationNet()
#best_model = MyOwnNetwork()

################################################################################
# TODO:                                                                        #
# Implement your own neural network and find suitable hyperparameters  #
# Be sure to edit the MyOwnNetwork class in the following code snippet #
# to upload the correct model!                                                 #
################################################################################

pass

################################################################################
#                             END OF YOUR CODE                                 #
################################################################################
```

Now it's time to edit the ranges above and adjust them to explore regions that performed well!

Also, feel free to experiment around! Also the network architecture, optimizer options and activations functions, etc. are hyperparameters that you can change!

Try to get your accuracy as high as possible! That's all what counts for this submission!

You'll pass if you reach at least **48%** accuracy on our test set - but there will also be a leaderboard of all students of this course. Will you make it to the top?

3.5 Checking the validation accuracy

In [52]:
```
labels, pred, acc = best_model.get_dataset_prediction(dataloaders['train'])
print("Train Accuracy: {}%".format(acc*100))
labels, pred, acc = best_model.get_dataset_prediction(dataloaders['val'])
print("Validation Accuracy: {}%".format(acc*100))
```

```
Train Accuracy: 10.106169871794872%
Validation Accuracy: 10.036057692307693%
```


4. Test your model

When you have finished your hyperparameter tuning and are sure you have your final model that performs well on the validation set (**you should at least get 48% accuracy on the validation set!**), it's time to run your model on the test set.

Important

As you have learned in the lecture, you must only use the test set one single time! So only run the next cell if you are really sure your model works well enough and that you want to submit. Your test set is different from the test set on our server, so results may vary. Nevertheless, you will have a reasonable close approximation about your performance if you only do a final evaluation on the test set.

If you are an external student that can't use our submission webpage: this test performance is your final result and if you surpassed the threshold, you have completed this exercise :). Now, train again to aim for a better number!

In [ ]:
```
# comment this part out to see your model's performance on the test set.
labels, pred, acc = best_model.get_dataset_prediction(dataloaders['test'])
print("Test Accuracy: {}%".format(acc*100))
```

Note:

The "real" test set is actually the dataset we're using for testing your model, which is **different** from the test-set you're using here.

5. Saving your Model

In [ ]:
```
from exercise_code.tests import save_pickle
save_pickle({"cifar_fcn": best_model}, "cifar_fcn.p")
```

In [ ]:
```
from exercise_code.submit import submit_exercise

submit_exercise('exercise06')
```

6. Submission Instructions

Congratulations! You've just built your first image classifier! To complete the exercise, submit your final model to our submission portal - you probably know the procedure by now.

1. Go on our submission page (https://i2dl.vc.in.tum.de/), register for an account and login. We use your matriculation number and send an email with the login details to the mail account associated. When in doubt, login into tum-online and check your mails there. You will get an id which we need in the next step.
2. Log into our submission page (https://i2dl.vc.in.tum.de/) with your account details and upload the zip file.
3. Your submission will be evaluated by our system and you will get feedback about the performance of it. You will get an email with your score as well as a message if you have surpassed the threshold.
4. Within the working period, you can submit as many solutions as you want to get the best possible score.

7. Submission Goals

- Goal: Successfully implement a fully connected NN image classifier, tune hyperparameters.
- Passing Criteria: This time, there are no unit tests that check specific components of your code. The only thing that's required to pass the submission, is your model to reach at least **48% accuracy** on **our** test dataset. The submission system will show you a number between 0 and 100 which corresponds to your accuracy.
- Submission start: **May 20, 2020 13:00:00**
- Submission deadline : **May 26, 2021 15:59:59**
- You can make ∞ submissions until the deadline. Your **best submission** will be considered for bonus

In [ ]: