Exercise Sheet 2 - CIFAR, MLP, Overfitting, and Regularization

- Deep Learning for Computer Vision – Winter Term 2022/23
- Organizers: Anwai Archit, Laura Hansel, Michaela Vystrcilova, Constantin Pape
- Tutors: Anwai Archit, Mai Elshazly, Andreas Schneider, Fabio Seel, Shashwat Sridhar
- Due date: **Monday, Nov 21, before 14:00**

Time required to solve this exercise sheet

As you will train a large number of models on this exercise sheet, model training will require an increased amount of time. So we recommend to start working on this sheet early.

Topic

In this exercise sheet, you will:

- get to know a new dataset: CIFAR-10
- implement a MLP
- get more familiar with model fitting
- see overfitting
- implement early stopping
- explore hyperparameters and their influence
- vary architecture to improve model performance

We are looking forward to seeing your solutions! Have fun!

IMPORTANT SUBMISSION INSTRUCTIONS

- **You need to answer all questions in written form!**
- When you've completed the exercise, download the notebook and rename it to `__.ipynb`
- Only submit the Jupyter Notebook (ipynb file). No other file is required. Upload it on `Stud.IP -> Deep Learning for Computer Vision -> Files -> Submission of Homework 2` .
- Make only one submission of the exercise per group.
- The deadline is strict
- In addition to the submissions, every member of your group should be prepared to present the exercise in the tutorials.

Implementation

- Do not change the cells which are marked as "DO NOT CHANGE", similarly write your solution into the marked cells.

Imports

In [2]:
```
import os
import fastprogress
import time

import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import torchvision
from torch.utils.data import DataLoader
```

GPU and Cuda checks

In [3]:
```
# DO NOT CHANGE
def get_device(cuda_preference=False):
    """Gets pytorch device object. If cuda_preference=True and
        cuda is available on your system, returns a cuda device.

    Args:
        cuda_preference: bool, default True
            Set to true if you would like to get a cuda device

    Returns: pytorch device object
            Pytorch device
    """

    print('cuda available:', torch.cuda.is_available(),
          '; cudnn available:', torch.backends.cudnn.is_available(),
          '; num devices:', torch.cuda.device_count(),
          ';mps available:', torch.backends.mps.is_built())

    use_cuda = False if not cuda_preference else torch.cuda.is_available()
    use_mps = False if cuda_preference else torch.backends.mps.is_available()

    if use_cuda: device = 'cuda:0'
    if use_mps:  device = 'mps'
    else: device = 'cpu'
    # device = torch.device(  else if use_mps: 'mps' else 'cpu')
    device_name = torch.cuda.get_device_name(device) if use_cuda else 'cpu/mps'
    print('Using device', device_name)
    return device
```

In [4]:
```
# DO NOT CHANGE
device = get_device()

# Get number of cpus to use for faster parallelized data loading
num_cpus = os.cpu_count()
print(num_cpus, 'CPUs available')
```

```
cuda available: False ; cudnn available: False ; num devices: 0 ;mps available: True
Using device cpu/mps
10 CPUs available
```

Recommendation: Use GPU or TPU for faster model training. Exercise Sheet 1 explains how to do that on Kaggle.

Load data

In [5]:
```
# DO NOT CHANGE
def grab_data(data_dir, num_cpus=1):
    """Downloads CIFAR10 train and test set, stores them on disk, computes mean
        and standard deviation per channel of trainset, normalizes the train set
        accordingly.

    Args:
        data_dir (str): Directory to store data
        num_cpus (int, optional): Number of cpus that should be used to
            preprocess data. Defaults to 1.

    Returns:
        CIFAR10, CIFAR10, float, float: Returns trainset and testset as
            torchvision CIFAR10 dataset objects. Returns mean and standard
            deviation used for normalization.
    """
    trainset = torchvision.datasets.CIFAR10(data_dir, train=True, download=True,
                                            transform=torchvision.transforms.ToTensor())

    # Get normalization transform
    num_samples = trainset.data.shape[0]
```

```
    trainloader = torch.utils.data.DataLoader(trainset, batch_size=num_samples,
                                              num_workers=num_cpus)
    imgs, _ = next(iter(trainloader))
    dataset_mean = torch.mean(imgs, dim=(0,2,3))
    dataset_std = torch.std(imgs, dim=(0,2,3))

    normalized_transform = torchvision.transforms.Compose([
        torchvision.transforms.ToTensor(),
        torchvision.transforms.Normalize(dataset_mean, dataset_std)
    ])

    # Load again, now normalized
    trainset = torchvision.datasets.CIFAR10(data_dir, download=True, train=True,
                                            transform=normalized_transform)
    # Apply the same transform, computed from the train-set, to the test-set
    # so both have a similar distribution. We do not normalize the test-set directly,
    # since we are not allowed to perform any computations with it. (We only use it
    # for reporting results in the very end)
    testset = torchvision.datasets.CIFAR10(data_dir, download=True, train=False,
                                           transform=normalized_transform)

    return trainset, testset, dataset_mean, dataset_std


def generate_train_val_data_split(trainset, split_seed=42, val_frac=0.2):
    """Splits train dataset into train and validation dataset.

    Args:
        trainset (CIFAR10): CIFAR10 trainset object
        split_seed (int, optional): Seed used to randomly assign data
            points to the validation set. Defaults to 42.
        val_frac (float, optional): Fraction of training set that should be
            split into validation set. Defaults to 0.2.

    Returns:
        CIFAR10, CIFAR10: CIFAR10 trainset and validation set.
    """
    num_val_samples = np.ceil(val_frac * trainset.data.shape[0]).astype(int)
    num_train_samples = trainset.data.shape[0] - num_val_samples
    trainset, valset = torch.utils.data.random_split(trainset,
                                (num_train_samples, num_val_samples),
                                generator=torch.Generator().manual_seed(split_seed))
    return trainset, valset


def init_data_loaders(trainset, valset, testset, batch_size=1024, num_cpus=0):
    """Initialize train, validation and test data loader.

    Args:
        trainset (CIFAR10): Training set torchvision dataset object.
        valset (CIFAR10): Validation set torchvision dataset object.
        testset (CIFAR10): Test set torchvision dataset object.
        batch_size (int, optional): Batchsize that should be generated by
            pytorch dataloader object. Defaults to 1024.
        num_cpus (int, optional): Number of CPUs to use when iterating over
            the data loader. More is faster. Defaults to 1.

    Returns:
        DataLoader, DataLoader, DataLoader: Returns pytorch DataLoader objects
            for training, validation and testing.
    """
    trainloader = torch.utils.data.DataLoader(trainset,
                                                   batch_size=batch_size,
                                                   shuffle=True,
                                                   num_workers=num_cpus)
    valloader = torch.utils.data.DataLoader(valset,
                                                 batch_size=batch_size,
                                                 shuffle=False,
                                                 num_workers=num_cpus)
    testloader = torch.utils.data.DataLoader(testset,
                                                  batch_size=batch_size,
                                                  shuffle=False,
                                                  num_workers=num_cpus)
    return trainloader, valloader, testloader
```

TODO

- Load the CIFAR 10 train and test data set using the functions defined above
- Generate a validation set from 20% of the training set samples. *Remember:* Keep the seed for the validation set generation fixed for reproducibility.
- Generate torch data loaders for the train, validation and test data set splits. Use a batch size of 1024.

Hint: we will use the mean and standard deviation returned by `grab_data` later

In [6]:

```
####################
cifar10_train, cifar10_test, cifar10_mean, cifar10_std = grab_data('/Users/simonblaue/ownCloud/Master/Deep Lear
####################
```

```
Files already downloaded and verified
Files already downloaded and verified
Files already downloaded and verified
```

In [7]:

```
trainset, valset = generate_train_val_data_split(cifar10_train)
trainloader, valloader, testloader = init_data_loaders(trainset, valset, cifar10_test)
```

Let's have a look at the dataset.

TODO

- Print all class names
- Plot 16 images randomly drawn from the training set with their according class label

Hint: Since you normalized the dataset before, you have to undo that operation for plotting

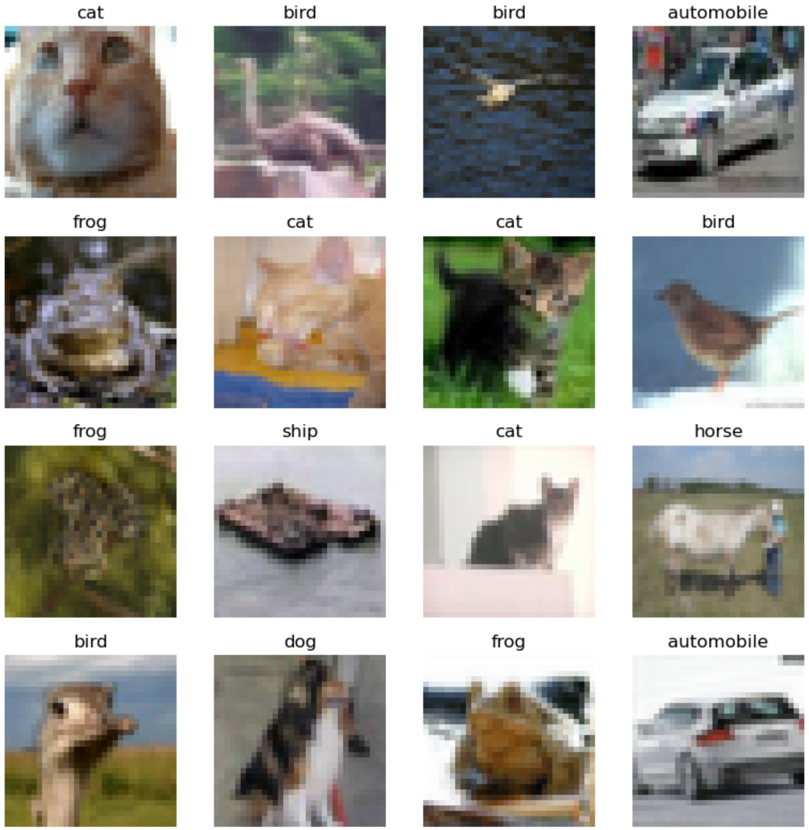
In [8]:

```
def imshow(img, mean, std,title=''):
    """Undo normalization using mean and standarddeviation and show image.

    Args:
        img (torch.Tensor): Image to show
        mean (np.array shape (3,)): Vector of means per channel used to
            normalize the dataset.
        std (np.array shape (3,)): Vector of standard deviations per channel
            used to normalize the dataset.
    """
    # Define function to plot

    ####################
    unnormalize = torchvision.transforms.Normalize((-mean / std).tolist(), (1.0 / std).tolist())
    img = unnormalize(img)
    plt.title(title)
    plt.imshow(img.permute(1,2,0))
    plt.axis('off')

    ####################
```

In [9]:

```
# Create actual plot and print the class names

###################
print("Class Names:", cifar10_train.classes)
print("---------------------------")
plt.figure(figsize=(8,8))
for i in range(16):
    img, label = cifar10_train[np.random.randint(0, len(cifar10_train))]
    plt.subplot(4,4,i+1)
    imshow(img, cifar10_mean, cifar10_std, title=cifar10_train.classes[label])
plt.tight_layout()
plt.show()
###################
```

```
Class Names: ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
---------------------------
```



Training, evaluation and plotting functions from Exercise 1

Here, we provide examples of the functions you implemented on the first exercise sheet to you. Some parts are still missing. You can ignore that for the time being, as you will implement that later as soon as the according functionality is required.

In [10]:

```
def accuracy(correct, total):
    """Compute accuracy as percentage.

    Args:
```

```
        correct (int): Number of samples correctly predicted.
        total (int): Total number of samples

    Returns:
        float: Accuracy
    """
    return float(correct)/total


def train(dataloader, optimizer, model, loss_fn, device, master_bar):
    """Run one training epoch.

    Args:
        dataloader (DataLoader): Torch DataLoader object to load data
        optimizer: Torch optimizer object
        model (nn.Module): Torch model to train
        loss_fn: Torch loss function
        device (torch.device): Torch device to use for training
        master_bar (fastprogress.master_bar): Will be iterated over for each
            epoch to draw batches and display training progress

    Returns:
        float, float: Mean loss of this epoch, fraction of correct predictions
            on training set (accuracy)
    """
    epoch_loss = []
    epoch_correct, epoch_total = 0, 0

    for x, y in fastprogress.progress_bar(dataloader, parent=master_bar):
        optimizer.zero_grad()
        model.train()

        # Forward pass
        y_pred = model(x.to(device))

        # For calculating the accuracy, save the number of correctly classified
        # images and the total number
        epoch_correct += sum(y.to(device) == y_pred.argmax(dim=1))
        epoch_total += len(y)

        # Compute loss
        loss = loss_fn(y_pred, y.to(device))

        # Backward pass
        loss.backward()
        optimizer.step()

        # For plotting the train loss, save it for each sample
        epoch_loss.append(loss.item())

    # Return the mean loss and the accuracy of this epoch
    return np.mean(epoch_loss), accuracy(epoch_correct, epoch_total)


def validate(dataloader, model, loss_fn, device, master_bar):
    """Compute loss, accuracy and confusion matrix on validation set.

    Args:
        dataloader (DataLoader): Torch DataLoader object to load data
        model (nn.Module): Torch model to train
        loss_fn: Torch loss function
        device (torch.device): Torch device to use for training
        master_bar (fastprogress.master_bar): Will be iterated over to draw
            batches and show validation progress

    Returns:
        float, float, torch.Tensor shape (10,10): Mean loss on validation set,
            fraction of correct predictions on validation set (accuracy)
    """
    epoch_loss = []
    epoch_correct, epoch_total = 0, 0
    confusion_matrix = torch.zeros(10, 10)

    model.eval()
    with torch.no_grad():
        for x, y in fastprogress.progress_bar(dataloader, parent=master_bar):
            # make a prediction on validation set
            y_pred = model(x.to(device))

            # For calculating the accuracy, save the number of correctly
```

```python
            # classified images and the total number
            epoch_correct += sum(y.to(device) == y_pred.argmax(dim=1))
            epoch_total += len(y)

            # Fill confusion matrix
            for (y_true, y_p) in zip(y, y_pred.argmax(dim=1)):
                confusion_matrix[int(y_true), int(y_p)] +=1

            # Compute loss
            loss = loss_fn(y_pred, y.to(device))

            # For plotting the train loss, save it for each sample
            epoch_loss.append(loss.item())

    # Return the mean loss, the accuracy and the confusion matrix
    return np.mean(epoch_loss), accuracy(epoch_correct, epoch_total), confusion_matrix


def run_training(model, optimizer, loss_function, device, num_epochs,
                train_dataloader, val_dataloader, early_stopper=None, verbose=False):
    """Run model training.

    Args:
        model (nn.Module): Torch model to train
        optimizer: Torch optimizer object
        loss_fn: Torch loss function for training
        device (torch.device): Torch device to use for training
        num_epochs (int): Max. number of epochs to train
        train_dataloader (DataLoader): Torch DataLoader object to load the
            training data
        val_dataloader (DataLoader): Torch DataLoader object to load the
            validation data
        early_stopper (EarlyStopper, optional): If passed, model will be trained
            with early stopping. Defaults to None.
        verbose (bool, optional): Print information about model training.
            Defaults to False.

    Returns:
        list, list, list, list, torch.Tensor shape (10,10): Return list of train
            losses, validation losses, train accuracies, validation accuracies
            per epoch and the confusion matrix evaluated in the last epoch.
    """
    start_time = time.time()
    master_bar = fastprogress.master_bar(range(num_epochs))
    train_losses, val_losses, train_accs, val_accs = [],[],[],[]

    for epoch in master_bar:
        # Train the model
        epoch_train_loss, epoch_train_acc = train(train_dataloader, optimizer, model,
                                                  loss_function, device, master_bar)
        # Validate the model
        epoch_val_loss, epoch_val_acc, confusion_matrix = validate(val_dataloader,
                                                                   model, loss_function,
                                                                   device, master_bar)

        # Save loss and acc for plotting
        train_losses.append(epoch_train_loss)
        val_losses.append(epoch_val_loss)
        train_accs.append(epoch_train_acc)
        val_accs.append(epoch_val_acc)

        if verbose:
            master_bar.write(f'Train loss: {epoch_train_loss:.2f}, val loss: {epoch_val_loss:.2f}, train acc: {

        if early_stopper:
            ####################
            early_stopper.new_acc = epoch_val_acc
            if early_stopper.early_stop:
                early_stopper.save(model)
                return train_losses, val_losses, train_accs, val_accs, confusion_matrix
            ####################
            #raise NotImplementedError # Comment out this keyword after your implementation

            # END OF YOUR CODE #

    time_elapsed = np.round(time.time() - start_time, 0).astype(int)
    print(f'Finished training after {time_elapsed} seconds.')
    return train_losses, val_losses, train_accs, val_accs, confusion_matrix
```

```python
def plot(title, label, train_results, val_results, yscale='linear', save_path=None,
         extra_pt=None, extra_pt_label=None):
    """Plot learning curves.

    Args:
        title (str): Title of plot
        label (str): x-axis label
        train_results (list): Results vector of training of length of number
            of epochs trained. Could be loss or accuracy.
        val_results (list): Results vector of validation of length of number
            of epochs. Could be loss or accuracy.
        yscale (str, optional): Matplotlib.pyplot.yscale parameter.
            Defaults to 'linear'.
        save_path (str, optional): If passed, figure will be saved at this path.
            Defaults to None.
        extra_pt (tuple, optional): Tuple of length 2, defining x and y coordinate
            of where an additional black dot will be plotted. Defaults to None.
        extra_pt_label (str, optional): Legend label of extra point. Defaults to None.
    """

    epoch_array = np.arange(len(train_results)) + 1
    train_label, val_label = "Training "+label.lower(), "Validation "+label.lower()

    sns.set(style='ticks')

    plt.plot(epoch_array, train_results, epoch_array, val_results, linestyle='dashed', marker='o')
    legend = ['Train results', 'Validation results']

    if extra_pt:
        ####################
        ## YOUR CODE HERE ##
        ####################
        raise NotImplementedError # Comment out this keyword after your implementation

        # END OF YOUR CODE #

    plt.legend(legend)
    plt.xlabel('Epoch')
    plt.ylabel(label)
    plt.yscale(yscale)
    plt.title(title)

    sns.despine(trim=True, offset=5)
    plt.title(title, fontsize=15)
    if save_path:
        plt.savefig(str(save_path), bbox_inches='tight')
    plt.show()
```

MLP model

TODO

- Define an MLP model implementing all the functionality indicated by the parameters and the docstrings
- There should be a non-linearity after the input layer and in the hidden layers, i.e. in all layers that map to hidden units, but not in the final (linear) layer that creates the outputs

Hint: As CIFAR 10 contains color images, amongst other dimensions you want to flatten the color channel dimension, too.

In [11]:
```
from collections import OrderedDict

class mlp(torch.nn.Module):
    """Multi layer perceptron torch model."""
    def __init__(self, img_width, num_in_channels, num_classes,
                 num_hidden_units=30, num_hidden_layers=1, act_fn=torch.nn.ReLU()):
        """ Initializes internal Module state. """

        super(mlp, self).__init__()

        layers = [
            ('Flatten', torch.nn.Flatten()),
            ('Input', torch.nn.Linear(img_width**2*num_in_channels, num_hidden_units)),
            ('Activation0',act_fn)
        ]

        for i in range(num_hidden_layers):
            layers.append((f'hidden{i}',torch.nn.Linear(num_hidden_units, num_hidden_units) ))
            layers.append((f'activation{i}', act_fn))

        layers.append(('Output',torch.nn.Linear(num_hidden_units, num_classes)))

        self.layers = nn.Sequential(
            OrderedDict(layers)
        )


    def forward(self, x):
        """ Defines the computation performed at every call. """
        # What are the dimensions of your input layer?

        for layer in self.layers:
            x = layer(x)
        return x
```

Model training: learning rate

One of the most important hyperparameters is the learning rate. If we set it incorrectly, our model might not train at all, take very long time to train, or lead to suboptimal performance. Thus, we should make sure to set it appropriately.

So with what learning rate should we start? Usually, you would start setting a very high learning rate, e.g. `lr = 1e0` and then decrease it by a factor of ten until the model starts to converge. Since we might have to try multiple values here, it is sufficient to train only a few iterations to see if the model trains at all. As soon as we find a learning rate that works, we train for more epochs to get a well performing model.

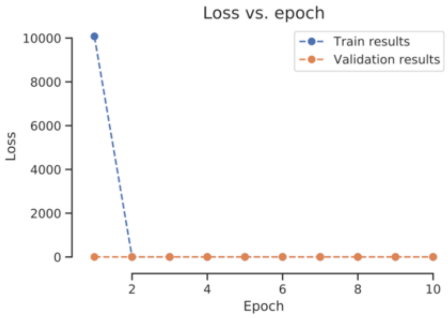
For optimization, we do not use SGD as in exercise 1, but the commonly used Adam optimizer, since it behaves more robustly and is easy to use.

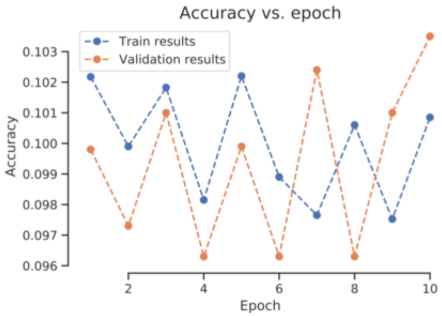
TODO

- Instantiate a MLP model with one hidden layer and ReLU activation function
- Train the model for 10 epochs
- Use the Adam optimizer
- Start with a learning rate of 10^0, then decrease the learning rate logarithmically, i.e. by a factor of 10, until your model starts to train
- Plot the training curves of the loss and the accuracies as in exercise 1. Use the functions defined above.

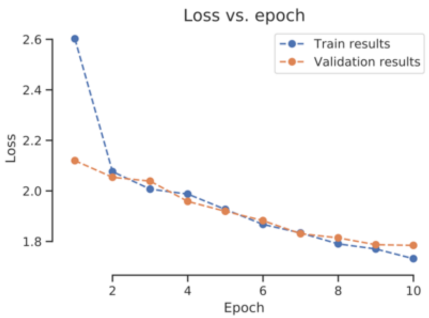
Hints:

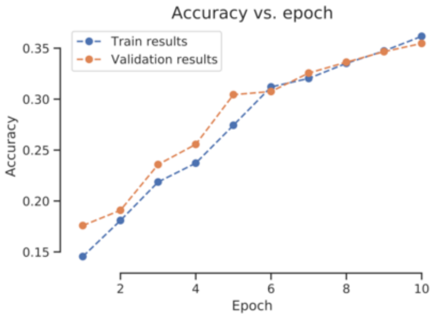
This is an example of a model that does not train sufficiently: *(Why?)*





This is an example of a model that does train: *(Why?)*





In [36]:

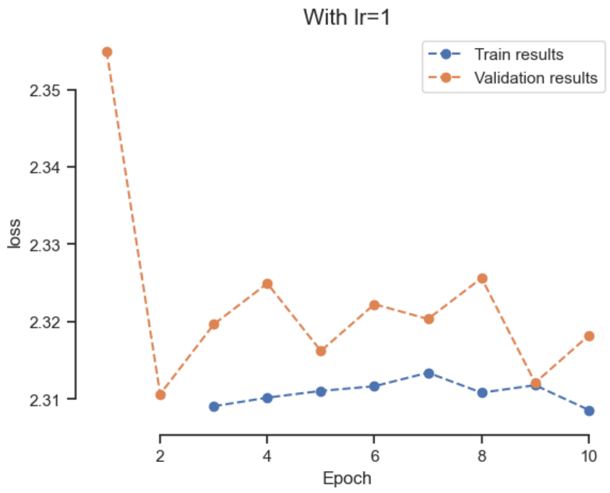
```
###################
img_width = 32
img_channels = 3
num_classes = 10
num_hidden_units = 30
num_hidden_layers = 1
learning_rate = 1
model = mlp(img_width=32, num_in_channels=3, num_classes=10, num_hidden_units=30, num_hidden_layers=1).to(devic

optimizer = optim.Adam(model.parameters(), lr=learning_rate)
loss_fn = torch.nn.CrossEntropyLoss()
print(model)
###################
```

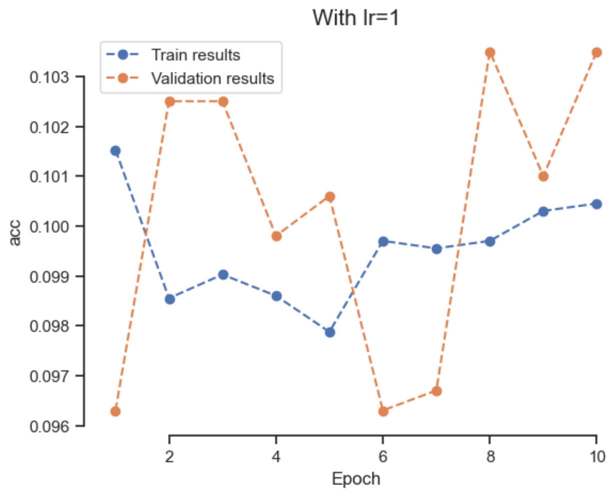
```
mlp(
  (layers): Sequential(
    (Flatten): Flatten(start_dim=1, end_dim=-1)
    (Input): Linear(in_features=3072, out_features=30, bias=True)
    (Activation0): ReLU()
    (hidden0): Linear(in_features=30, out_features=30, bias=True)
    (activation0): ReLU()
    (Output): Linear(in_features=30, out_features=10, bias=True)
  )
)
```

In [49]:

```
lr = 1
model = mlp(img_width=32, num_in_channels=3, num_classes=10, num_hidden_units=30, num_hidden_layers=1).to(devic
optimizer = optim.Adam(model.parameters(), lr=lr)
train_losses1, val_losses1, train_accs1, val_accs1, confusion_matrix1 = run_training(model=model, optimizer=opt
plot(title=f'With lr={lr}', label='loss', train_results=train_losses1, val_results=val_losses1)
plot(title=f'With lr={lr}', label='acc', train_results=train_accs1, val_results=val_accs1)
plt.show()
```

```
Finished training after 89 seconds.
```

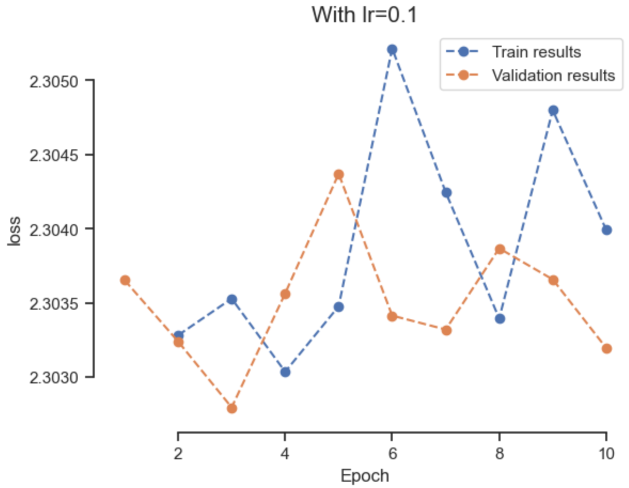


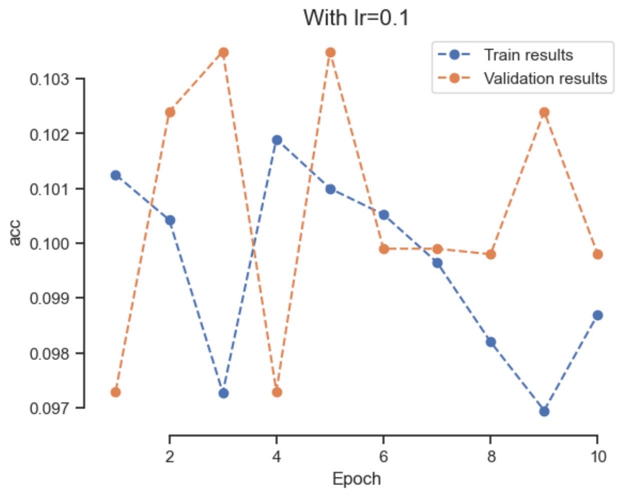


In [50]:

```
lr = 0.1
model = mlp(img_width=32, num_in_channels=3, num_classes=10, num_hidden_units=30, num_hidden_layers=1).to(devic
optimizer = optim.Adam(model.parameters(), lr=lr)
train_losses01, val_losses01, train_accs01, val_accs01, confusion_matrix01 = run_training(model=model, optimize
plot(title=f'With lr={lr}', label='loss', train_results=train_losses01, val_results=val_losses01)
plot(title=f'With lr={lr}', label='acc', train_results=train_accs01, val_results=val_accs01)
plt.show()
```

```
Finished training after 86 seconds.
```

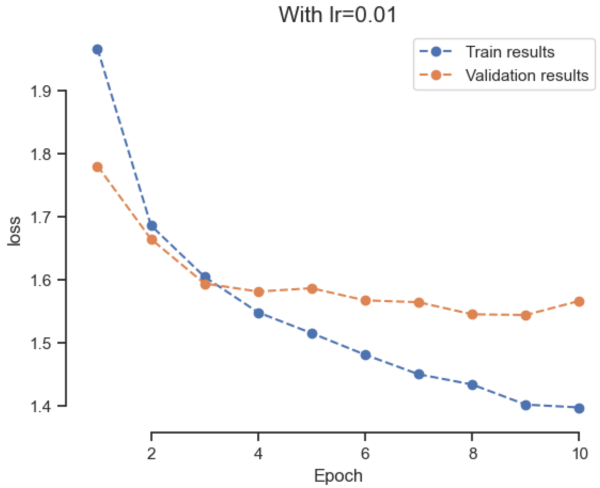


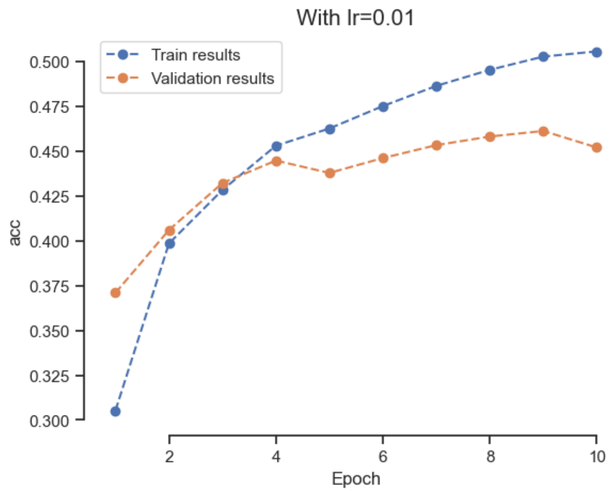


In [51]:

```
lr = 0.01
model = mlp(img_width=32, num_in_channels=3, num_classes=10, num_hidden_units=30, num_hidden_layers=1).to(devic
optimizer = optim.Adam(model.parameters(), lr=lr)
train_losses001, val_losses001, train_accs001, val_accs001, confusion_matrix001 = run_training(model=model, opt
plot(title=f'With lr={lr}', label='loss', train_results=train_losses001, val_results=val_losses001)
plot(title=f'With lr={lr}', label='acc', train_results=train_accs001, val_results=val_accs001)
plt.show()
```

```
Finished training after 86 seconds.
```





Has your model already converged, i.e. reached the highest accuracy on the validation set? Probably not. So here are your todo's:

TODO:

- Train the model for 100 epochs (this might take approx. 30 min depending on your GPU)

TODO from now on, for all subsequent tasks:

- Print the overall best value and the epoch at which it occurred of:
  - val loss and
  - val accuracy

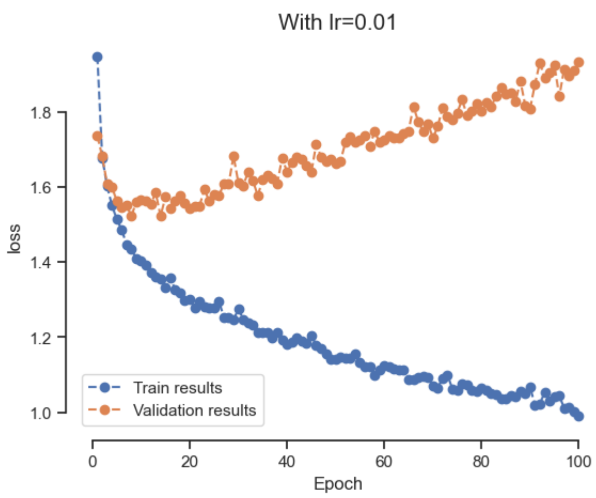
In [15]:

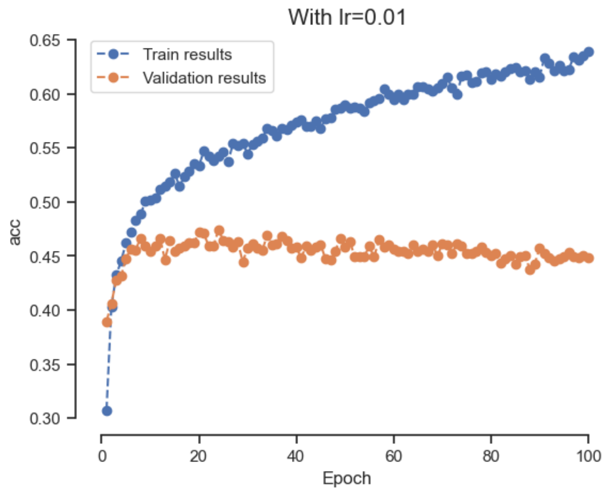
```
def print_best(val_losses,val_accs):
  print(f"Minimal validation lost: {min(val_losses)}, occured at epoch: {val_losses.index(min(val_losses))}")
  print(f"Maximal accuracy: {max(val_accs)}, occured at epoch: {val_accs.index(max(val_accs))}")
```

In [54]:

```
####################
lr = 0.01
model = mlp(img_width=32, num_in_channels=3, num_classes=10, num_hidden_units=30, num_hidden_layers=1).to(devic
optimizer = optim.Adam(model.parameters(), lr=lr)
train_losses, val_losses, train_accs, val_accs, confusion_matrix = run_training(model=model, optimizer=optimize
plot(title=f'With lr={lr}', label='loss', train_results=train_losses, val_results=val_losses)
plot(title=f'With lr={lr}', label='acc', train_results=train_accs, val_results=val_accs)
plt.show()
####################
```

```
Finished training after 857 seconds.
```





In [69]:

```
print_best(val_losses, val_accs)
```

```
Minimal validation lost: 1.522628915309906, occured at epoch: 13
Maximal accuracy: 0.4744, occured at epoch: 23
```

Let's have a look at those training curves! Here are some questions for you.

TODO

Answer the following questions in written form, as they are really crucial for the rest of this course.

1. Does the training loss decrease after each epoch? Why does it? // Why does it not?
2. Does the validation loss decrease after each epoch? Why does it? // Why does it not? (For your answer to be sufficient, you should describe fluctuations and discuss the overall minimum of the curve.)
3. Do the training and validation accuracy increase after each epoch? Why? // Why not?
4. Are the epochs at which you got the best validation loss and the best validation accuracy the same?
5. (Optional): Do you have any ideas why not?
6. At which epoch was your model best? I.e. if you had saved your model after each training epoch, which one would you use to make predictions to unseen samples (e.g. from the test set)? Why? (For your answer to be sufficient: Also discuss what this means in terms of overfitting)

Your answers:

1. No, fluctuations, but overall decreasing -> because possible to "overfitting" or fitt better and better
2. No, fluctuations -> when overfitted rises again
3. Again, no same as above
4. No
5.
6. Best means highest accuracy on validation -> epoch 23

Save and restore model checkpoints

Training that model for 100 epochs took quite a bit of time, right? Wouldn't it be a pity if it would get deleted out of memory, e.g. because your Colab session terminates (this can even happen automatically)? We would have to train it again to make predictions! To prevent this, we would like to save a check-point of the already optimized model's weights to disk. Then, we could just load our model weights at any time and use our model again without retraining. As you will see in a bit, this will be very handy for early stopping, too!

TODO

- Save a checkpoint of the `model` trained above (i.e. the model's parameters) to disk
- Initialize a new model, `model2` with the same architecture as used for the `model` you stored. Do *not* train `model2` .
- Compute `model2` 's validation set accuracy. *Hint:* You can use the validation function from above. As a parameter, you would have to set `master_bar=None` since there is no progress bar for epochs in this setting.
- Now, overwrite the initialized, untrained weights of `model2` with the weights you saved into the checkpoint of `model` .
- Evaluate `model2` 's validation set accuracy again. It should be of the exact same value as `model` 's validation set accuracy.

Hints:

- Read https://pytorch.org/tutorials/beginner/saving_loading_models.html
- Use `model.state_dict()`

In [55]:
```
####################
save_path = '/Users/simonblaue/ownCloud/Master/Deep Learning for Computer Vision/Exercise2/saves/model'
torch.save(model, save_path)

model2 = torch.load(save_path)
####################
```

In [60]:
```
val_loss_m, val_acc_m, confusion_matrix_m = validate(valloader, model, loss_fn, device, master_bar=None)
val_loss_m2, val_acc_m2, confusion_matrix_m2 = validate(valloader, model2, loss_fn, device, master_bar=None)
```

100.00% [10/10 00:03<00:00]

100.00% [10/10 00:03<00:00]

In [61]:
```
print(val_loss_m, val_acc_m)
print(val_loss_m2, val_acc_m2)
```

```
1.93424232006073 0.4489
1.93424232006073 0.4489
```

Early stopping

So the model you ended up with after 100 epochs was not the best one. That has two implications for us: (1) We would not have had to train for that many epochs and could have saved some computing time. (2) We do not have the best model to apply our model to make actual predictions for unseen samples. If we would constantly assess our model's validation performance during training, we could stop optimization as soon as our model's performance does not increase anymore. This is called *early stopping*.

TODO:

- Implement the `EarlyStopper` class below
- Modify `def run_training(...)` above such that it updates the `EarlyStopper` after each training epoch. Stop training as soon as the validation accuracy did not increase anymore. Then, load the model checkpoint of the previous epoch (i.e. your best model)

In [24]:
```
class EarlyStopper:
    """Early stops the training if validation accuracy does not increase after a
    given patience. Saves and loads model checkpoints.
    """
```

```
    def __init__(self, patience_number=5, verbose=False, path='checkpoint.pt'):
        """Initialization.

        Args:
            verbose (bool, optional): Print additional information. Defaults to False.
            path (str, optional): Path where checkpoints should be saved.
                Defaults to 'checkpoint.pt'.
        """
        ####################
        self.verbose = verbose
        self.path = path
        self._patience = patience_number
        self.accs = np.zeros(patience_number)
        self.new_acc = 0
        ####################

    @property
    def early_stop(self):
        """True if early stopping criterion is reached.

        Returns:
            [bool]: True if early stopping criterion is reached.
        """
        ####################
        if self.verbose: print("Checking acc:",self.accs[1:], '<', self.accs[0])
        for i,newer_acc in enumerate(self.accs[1:]):
            if newer_acc >= self.accs[0]:
                self.accs = np.roll(self.accs, -1)
                self.accs[-1] = self.new_acc
                return False

        return True
        ####################


    ####################
    def save(self, model):
        torch.save(model, self.path)
    ####################
    # define more methods required to make `EarlyStopper` functional
```

TODO:

- Train a MLP model (same architecture, optimization, etc. as you used before)
- Set `num_epochs = 100`
- Use your `EarlyStopper` to stop training early, after validation accuracy did not increase for one epoch (see description in TODOs above)

TODO here and for all subsequent exercises:

- In the training plots you create, mark the validation accuracy point of the model you end up with after stopping your training early. To do so, you can implement the missing functionality in `def plot(...)` above.

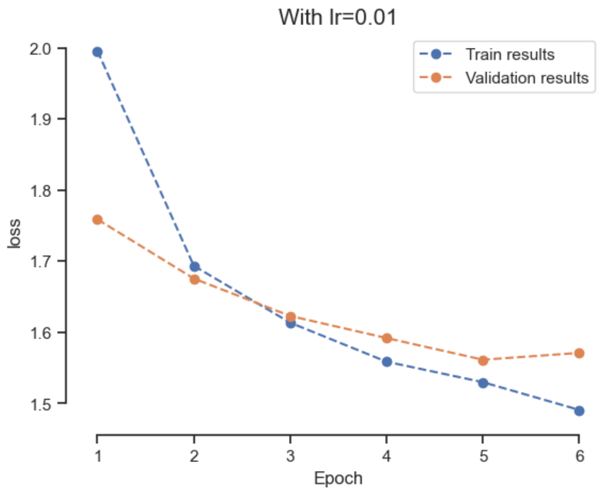
In [76]:

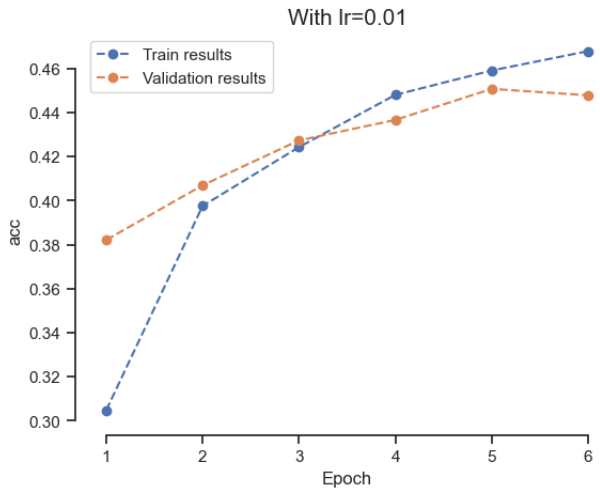
```
####################
lr = 0.01
model = mlp(img_width=32, num_in_channels=3, num_classes=10, num_hidden_units=30, num_hidden_layers=1).to(devic
optimizer = optim.Adam(model.parameters(), lr=lr)
train_losses_es, val_losses_es, train_accs_es, val_accs_es, confusion_matrix_es = run_training(model=model, opt
plot(title=f'With lr={lr}', label='loss', train_results=train_losses_es, val_results=val_losses_es)
plot(title=f'With lr={lr}', label='acc', train_results=train_accs_es, val_results=val_accs_es)
plt.show()
####################
```

5.00% [5/100 00:42<13:35]

100.00% [10/10 00:03<00:00]

```
Checking acc: 0 > 0.3822
Checking acc: 0.3822 > 0.407
Checking acc: 0.407 > 0.4274
Checking acc: 0.4274 > 0.4367
Checking acc: 0.4367 > 0.4508
Checking acc: 0.4508 > 0.4479
```


With lr=0.01
Train results
Validation results
loss
Epoch
2.0
1.9
1.8
1.7
1.6
1.5
1
2
3
4
5
6


With lr=0.01
Train results
Validation results
acc
Epoch
0.46
0.44
0.42
0.40
0.38
0.36
0.34
0.32
0.30
1
2
3
4
5
6

TODO

- Compare the training you just did with the one of the same model trained for 100 epochs. Did you reach best model performance? If so: why? If not: why not?
- Implement a patience functionality into `EarlyStopper` : stop model training, if validation accuracy did not increase for `patience` epochs. You are allowed to add more arguments to `EarlyStopper.__init__` .
- Do the same training as in the previous cell, starting training from scratch, but try different values for `patience` now. Did you end up with a model resulting in the best validation accuracy you have seen so far, but without training the full 100 epochs?

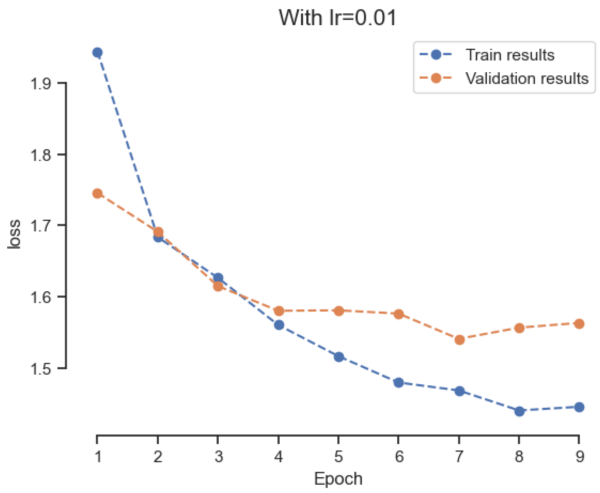
In [97]:

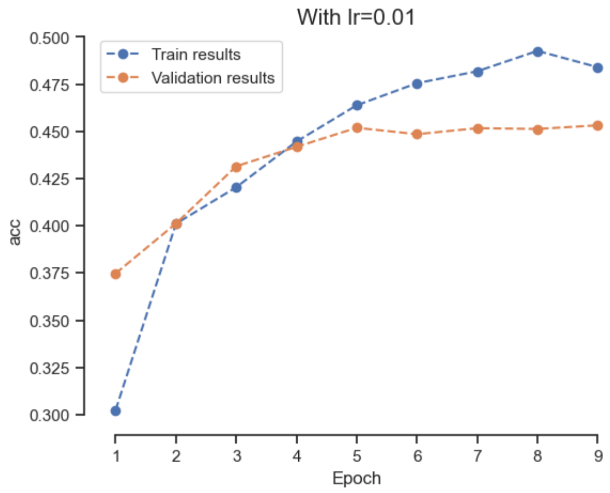
```
####################
lr = 0.01
patience = 3
save_path = 'saves/model_pt3.pt'
model = mlp(img_width=32, num_in_channels=3, num_classes=10, num_hidden_units=30, num_hidden_layers=1).to(devic
optimizer = optim.Adam(model.parameters(), lr=lr)
train_losses_es_pt3, val_losses_es_pt3, train_accs_es_pt3, val_accs_es_pt3, confusion_matrix_es_pt3 = run_train
plot(title=f'With lr={lr}', label='loss', train_results=train_losses_es_pt3, val_results=val_losses_es_pt3)
plot(title=f'With lr={lr}', label='acc', train_results=train_accs_es_pt3, val_results=val_accs_es_pt3)
plt.show()
print_best(val_losses_es_pt3, val_accs_es_pt3)
####################
```

8.00% [8/100 01:10<13:25]

100.00% [10/10 00:03<00:00]

```
Checking acc: [0. 0. 0.] < 0.0
Checking acc: [0.     0.     0.3748] < 0.0
Checking acc: [0.     0.3748 0.4012] < 0.0
Checking acc: [0.3748 0.4012 0.4315] < 0.0
Checking acc: [0.4012 0.4315 0.4418] < 0.3748
Checking acc: [0.4315 0.4418 0.4519] < 0.4012
Checking acc: [0.4418 0.4519 0.4485] < 0.4315
Checking acc: [0.4519 0.4485 0.4517] < 0.4418
Checking acc: [0.4485 0.4517 0.4513] < 0.4519
```






```
Minimal validation lost: 1.5409655570983887, occured at epoch: 6
Maximal accuracy: 0.4532, occured at epoch: 8
```

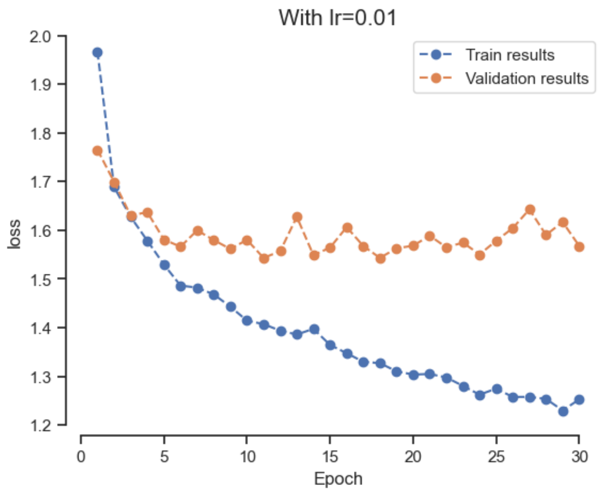
In [98]:

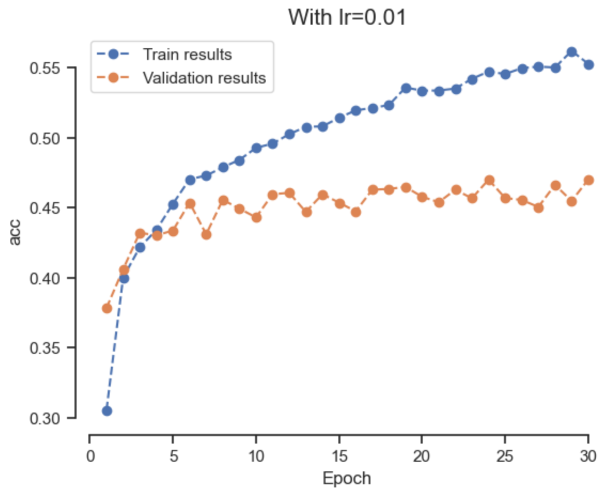
```
lr = 0.01
patience = 5
save_path = 'saves/model_pt5.pt'
model = mlp(img_width=32, num_in_channels=3, num_classes=10, num_hidden_units=30, num_hidden_layers=1).to(devic
optimizer = optim.Adam(model.parameters(), lr=lr)
train_losses_es_pt5, val_losses_es_pt5, train_accs_es_pt5, val_accs_es_pt5, confusion_matrix_es_pt5 = run_train
plot(title=f'With lr={lr}', label='loss', train_results=train_losses_es_pt5, val_results=val_losses_es_pt5)
plot(title=f'With lr={lr}', label='acc', train_results=train_accs_es_pt5, val_results=val_accs_es_pt5)
plt.show()
print_best(val_losses_es_pt5, val_accs_es_pt5)
```

29.00% [29/100 04:12<10:17]

100.00% [10/10 00:03<00:00]

```
Checking acc: [0. 0. 0. 0. 0.] < 0.0
Checking acc: [0.     0.     0.     0.     0.3783] < 0.0
Checking acc: [0.     0.     0.     0.3783 0.4059] < 0.0
Checking acc: [0.     0.     0.3783 0.4059 0.432 ] < 0.0
Checking acc: [0.     0.3783 0.4059 0.432  0.4301] < 0.0
Checking acc: [0.3783 0.4059 0.432  0.4301 0.4336] < 0.0
Checking acc: [0.4059 0.432  0.4301 0.4336 0.4535] < 0.3783
Checking acc: [0.432  0.4301 0.4336 0.4535 0.4309] < 0.4059
Checking acc: [0.4301 0.4336 0.4535 0.4309 0.4555] < 0.432
Checking acc: [0.4336 0.4535 0.4309 0.4555 0.4492] < 0.4301
Checking acc: [0.4535 0.4309 0.4555 0.4492 0.443 ] < 0.4336
Checking acc: [0.4309 0.4555 0.4492 0.443  0.4595] < 0.4535
Checking acc: [0.4555 0.4492 0.443  0.4595 0.4605] < 0.4309
Checking acc: [0.4492 0.443  0.4595 0.4605 0.4469] < 0.4555
Checking acc: [0.443  0.4595 0.4605 0.4469 0.4595] < 0.4492
Checking acc: [0.4595 0.4605 0.4469 0.4595 0.4535] < 0.443
Checking acc: [0.4605 0.4469 0.4595 0.4535 0.4473] < 0.4595
Checking acc: [0.4469 0.4595 0.4535 0.4473 0.4627] < 0.4605
Checking acc: [0.4595 0.4535 0.4473 0.4627 0.4633] < 0.4469
Checking acc: [0.4535 0.4473 0.4627 0.4633 0.4647] < 0.4595
Checking acc: [0.4473 0.4627 0.4633 0.4647 0.4577] < 0.4535
Checking acc: [0.4627 0.4633 0.4647 0.4577 0.4538] < 0.4473
Checking acc: [0.4633 0.4647 0.4577 0.4538 0.4627] < 0.4627
Checking acc: [0.4647 0.4577 0.4538 0.4627 0.4566] < 0.4633
Checking acc: [0.4577 0.4538 0.4627 0.4566 0.4699] < 0.4647
Checking acc: [0.4538 0.4627 0.4566 0.4699 0.4568] < 0.4577
Checking acc: [0.4627 0.4566 0.4699 0.4568 0.4554] < 0.4538
Checking acc: [0.4566 0.4699 0.4568 0.4554 0.4504] < 0.4627
Checking acc: [0.4699 0.4568 0.4554 0.4504 0.4662] < 0.4566
Checking acc: [0.4568 0.4554 0.4504 0.4662 0.4547] < 0.4699
```






```
Minimal validation lost: 1.543161654472351, occured at epoch: 10
Maximal accuracy: 0.4699, occured at epoch: 23
```

Which learning rate is best?

Now that we have a learning strategy that works well, let us explore the effect of the learning rate on training and model performance.

TODO:

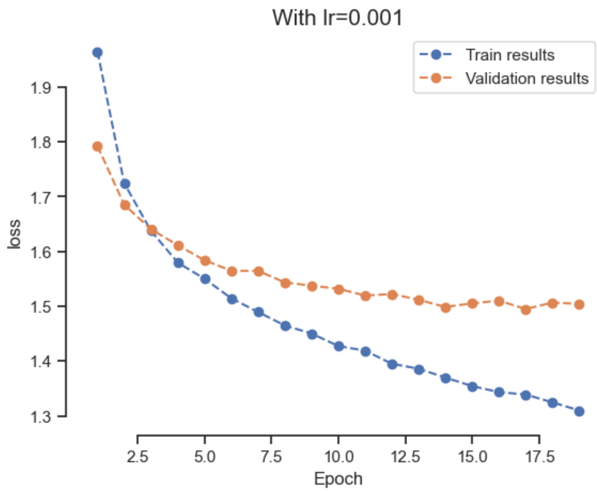
- Run training again as above, but with learning rate decreased by one order of magnitude, i.e. `lr = 1e-3`
- Run training again as above, but now with even smaller learning rate, `lr = 1e-4`
- What do you observe in terms of model accuracy? How long did it take to train these models? Which learning rate would you choose for any subsequent experiments you could do?

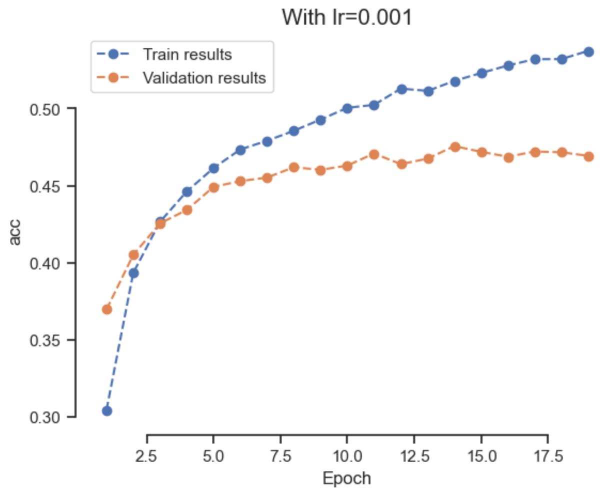
In [99]:

```
lr = 1e-3

####################
patience = 4
save_path = 'saves/model_lr_1e-3.pt'
model = mlp(img_width=32, num_in_channels=3, num_classes=10, num_hidden_units=30, num_hidden_layers=1).to(devic
optimizer = optim.Adam(model.parameters(), lr=lr)
train_losses_lr_s, val_losses_lr_s, train_accs_lr_s, val_accs_lr_s, confusion_matrix_lr_s = run_training(model=
plot(title=f'With lr={lr}', label='loss', train_results=train_losses_lr_s, val_results=val_losses_lr_s)
plot(title=f'With lr={lr}', label='acc', train_results=train_accs_lr_s, val_results=val_accs_lr_s)
plt.show()
print_best(val_losses_lr_s, val_accs_lr_s)
####################
```

18.00% [18/100 02:36<11:53]
100.00% [10/10 00:03<00:00]

```
Checking acc: [0. 0. 0. 0.] < 0.0
Checking acc: [0.     0.     0.     0.3697] < 0.0
Checking acc: [0.     0.     0.3697 0.4051] < 0.0
Checking acc: [0.     0.3697 0.4051 0.4255] < 0.0
Checking acc: [0.3697 0.4051 0.4255 0.4342] < 0.0
Checking acc: [0.4051 0.4255 0.4342 0.4492] < 0.3697
Checking acc: [0.4255 0.4342 0.4492 0.4529] < 0.4051
Checking acc: [0.4342 0.4492 0.4529 0.4552] < 0.4255
Checking acc: [0.4492 0.4529 0.4552 0.4619] < 0.4342
Checking acc: [0.4529 0.4552 0.4619 0.4602] < 0.4492
Checking acc: [0.4552 0.4619 0.4602 0.463 ] < 0.4529
Checking acc: [0.4619 0.4602 0.463  0.4708] < 0.4552
Checking acc: [0.4602 0.463  0.4708 0.4639] < 0.4619
Checking acc: [0.463  0.4708 0.4639 0.4675] < 0.4602
Checking acc: [0.4708 0.4639 0.4675 0.4756] < 0.463
Checking acc: [0.4639 0.4675 0.4756 0.4719] < 0.4708
Checking acc: [0.4675 0.4756 0.4719 0.4687] < 0.4639
Checking acc: [0.4756 0.4719 0.4687 0.4721] < 0.4675
Checking acc: [0.4719 0.4687 0.4721 0.4716] < 0.4756
```






```
Minimal validation lost: 1.4949013948440553, occured at epoch: 16
Maximal accuracy: 0.4756, occured at epoch: 13
```

In [100…

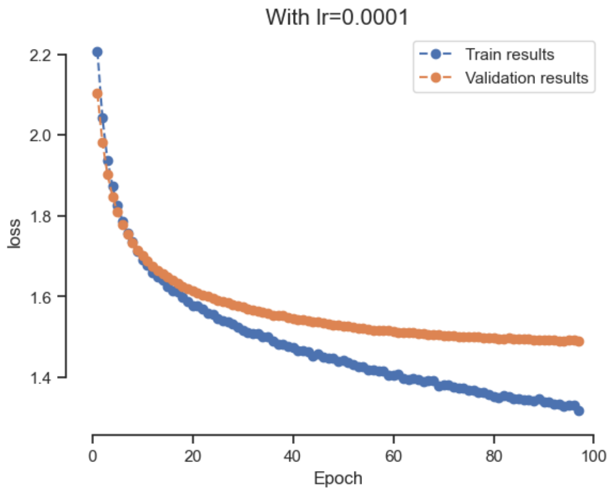
```
lr = 1e-4

####################
patience = 4
save_path = 'saves/model_lr_1e-4.pt'
model = mlp(img_width=32, num_in_channels=3, num_classes=10, num_hidden_units=30, num_hidden_layers=1).to(devic
optimizer = optim.Adam(model.parameters(), lr=lr)
train_losses_lr_xs, val_losses_lr_xs, train_accs_lr_xs, val_accs_lr_xs, confusion_matrix_lr_xs = run_training(m
plot(title=f'With lr={lr}', label='loss', train_results=train_losses_lr_xs, val_results=val_losses_lr_xs)
plot(title=f'With lr={lr}', label='acc', train_results=train_accs_lr_xs, val_results=val_accs_lr_xs)
plt.show()
print_best(val_losses_lr_xs, val_accs_lr_xs)
####################
```

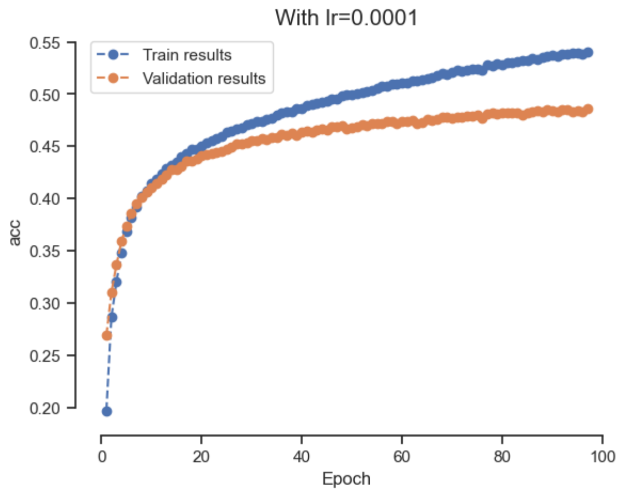
96.00% [96/100 13:55<00:34]

100.00% [10/10 00:03<00:00]

```
Checking acc: [0. 0. 0. 0.] < 0.0
Checking acc: [0.     0.     0.     0.2689] < 0.0
Checking acc: [0.     0.     0.2689 0.3099] < 0.0
Checking acc: [0.     0.2689 0.3099 0.3363] < 0.0
Checking acc: [0.2689 0.3099 0.3363 0.359 ] < 0.0
Checking acc: [0.3099 0.3363 0.359  0.3739] < 0.2689
Checking acc: [0.3363 0.359  0.3739 0.3855] < 0.3099
Checking acc: [0.359  0.3739 0.3855 0.3951] < 0.3363
Checking acc: [0.3739 0.3855 0.3951 0.4017] < 0.359
Checking acc: [0.3855 0.3951 0.4017 0.4063] < 0.3739
Checking acc: [0.3951 0.4017 0.4063 0.4108] < 0.3855
Checking acc: [0.4017 0.4063 0.4108 0.4147] < 0.3951
Checking acc: [0.4063 0.4108 0.4147 0.4188] < 0.4017
Checking acc: [0.4108 0.4147 0.4188 0.4232] < 0.4063
Checking acc: [0.4147 0.4188 0.4232 0.4283] < 0.4108
Checking acc: [0.4188 0.4232 0.4283 0.4279] < 0.4147
Checking acc: [0.4232 0.4283 0.4279 0.4312] < 0.4188
Checking acc: [0.4283 0.4279 0.4312 0.4362] < 0.4232
Checking acc: [0.4279 0.4312 0.4362 0.436 ] < 0.4283
Checking acc: [0.4312 0.4362 0.436  0.4385] < 0.4279
Checking acc: [0.4362 0.436  0.4385 0.4415] < 0.4312
```

```
Checking acc: [0.436  0.4385 0.4415 0.4426] < 0.4362
Checking acc: [0.4385 0.4415 0.4426 0.4431] < 0.436
Checking acc: [0.4415 0.4426 0.4431 0.4447] < 0.4385
Checking acc: [0.4426 0.4431 0.4447 0.4457] < 0.4415
Checking acc: [0.4431 0.4447 0.4457 0.447 ] < 0.4426
Checking acc: [0.4447 0.4457 0.447  0.4495] < 0.4431
Checking acc: [0.4457 0.447  0.4495 0.452 ] < 0.4447
Checking acc: [0.447  0.4495 0.452  0.4521] < 0.4457
Checking acc: [0.4495 0.452  0.4521 0.4531] < 0.447
Checking acc: [0.452  0.4521 0.4531 0.4556] < 0.4495
Checking acc: [0.4521 0.4531 0.4556 0.4559] < 0.452
Checking acc: [0.4531 0.4556 0.4559 0.4571] < 0.4521
Checking acc: [0.4556 0.4559 0.4571 0.4563] < 0.4531
Checking acc: [0.4559 0.4571 0.4563 0.4583] < 0.4556
Checking acc: [0.4571 0.4563 0.4583 0.4588] < 0.4559
Checking acc: [0.4563 0.4583 0.4588 0.4617] < 0.4571
Checking acc: [0.4583 0.4588 0.4617 0.4606] < 0.4563
Checking acc: [0.4588 0.4617 0.4606 0.4627] < 0.4583
Checking acc: [0.4617 0.4606 0.4627 0.461 ] < 0.4588
Checking acc: [0.4606 0.4627 0.461  0.4637] < 0.4617
Checking acc: [0.4627 0.461  0.4637 0.465 ] < 0.4606
Checking acc: [0.461  0.4637 0.465  0.4633] < 0.4627
Checking acc: [0.4637 0.465  0.4633 0.4655] < 0.461
Checking acc: [0.465  0.4633 0.4655 0.4668] < 0.4637
Checking acc: [0.4633 0.4655 0.4668 0.4657] < 0.465
Checking acc: [0.4655 0.4668 0.4657 0.4689] < 0.4633
Checking acc: [0.4668 0.4657 0.4689 0.4683] < 0.4655
Checking acc: [0.4657 0.4689 0.4683 0.4701] < 0.4668
Checking acc: [0.4689 0.4683 0.4701 0.4672] < 0.4657
Checking acc: [0.4683 0.4701 0.4672 0.468 ] < 0.4689
Checking acc: [0.4701 0.4672 0.468  0.4684] < 0.4683
Checking acc: [0.4672 0.468  0.4684 0.4707] < 0.4701
Checking acc: [0.468  0.4684 0.4707 0.47  ] < 0.4672
Checking acc: [0.4684 0.4707 0.47   0.4723] < 0.468
Checking acc: [0.4707 0.47   0.4723 0.4722] < 0.4684
Checking acc: [0.47   0.4723 0.4722 0.4719] < 0.4707
Checking acc: [0.4723 0.4722 0.4719 0.4736] < 0.47
Checking acc: [0.4722 0.4719 0.4736 0.4738] < 0.4723
Checking acc: [0.4719 0.4736 0.4738 0.4715] < 0.4722
Checking acc: [0.4736 0.4738 0.4715 0.4744] < 0.4719
Checking acc: [0.4738 0.4715 0.4744 0.4736] < 0.4736
Checking acc: [0.4715 0.4744 0.4736 0.4745] < 0.4738
Checking acc: [0.4744 0.4736 0.4745 0.4716] < 0.4715
Checking acc: [0.4736 0.4745 0.4716 0.4726] < 0.4744
Checking acc: [0.4745 0.4716 0.4726 0.4757] < 0.4736
Checking acc: [0.4716 0.4726 0.4757 0.4746] < 0.4745
Checking acc: [0.4726 0.4757 0.4746 0.476 ] < 0.4716
Checking acc: [0.4757 0.4746 0.476  0.4783] < 0.4726
Checking acc: [0.4746 0.476  0.4783 0.4784] < 0.4757
Checking acc: [0.476  0.4783 0.4784 0.4771] < 0.4746
Checking acc: [0.4783 0.4784 0.4771 0.4778] < 0.476
Checking acc: [0.4784 0.4771 0.4778 0.478 ] < 0.4783
Checking acc: [0.4771 0.4778 0.478  0.4787] < 0.4784
Checking acc: [0.4778 0.478  0.4787 0.4788] < 0.4771
Checking acc: [0.478  0.4787 0.4788 0.4802] < 0.4778
Checking acc: [0.4787 0.4788 0.4802 0.4769] < 0.478
Checking acc: [0.4788 0.4802 0.4769 0.4812] < 0.4787
Checking acc: [0.4802 0.4769 0.4812 0.4818] < 0.4788
Checking acc: [0.4769 0.4812 0.4818 0.4815] < 0.4802
Checking acc: [0.4812 0.4818 0.4815 0.4819] < 0.4769
Checking acc: [0.4818 0.4815 0.4819 0.4819] < 0.4812
Checking acc: [0.4815 0.4819 0.4819 0.4824] < 0.4818
Checking acc: [0.4819 0.4819 0.4824 0.4818] < 0.4815
Checking acc: [0.4819 0.4824 0.4818 0.4805] < 0.4819
Checking acc: [0.4824 0.4818 0.4805 0.4823] < 0.4819
Checking acc: [0.4818 0.4805 0.4823 0.4831] < 0.4824
Checking acc: [0.4805 0.4823 0.4831 0.4837] < 0.4818
Checking acc: [0.4823 0.4831 0.4837 0.483 ] < 0.4805
Checking acc: [0.4831 0.4837 0.483  0.4848] < 0.4823
Checking acc: [0.4837 0.483  0.4848 0.4844] < 0.4831
Checking acc: [0.483  0.4848 0.4844 0.4833] < 0.4837
Checking acc: [0.4848 0.4844 0.4833 0.4851] < 0.483
Checking acc: [0.4844 0.4833 0.4851 0.4849] < 0.4848
Checking acc: [0.4833 0.4851 0.4849 0.4836] < 0.4844
Checking acc: [0.4851 0.4849 0.4836 0.484 ] < 0.4833
Checking acc: [0.4849 0.4836 0.484  0.4834] < 0.4851
```






```
Minimal validation lost: 1.4907478213310241, occured at epoch: 92
Maximal accuracy: 0.4857, occured at epoch: 96
```

Explore batch size *(optional)*

This task is optional, you do not need to solve it

Let us explore even more model and training parameters. In this section, we will see the impact of batch size on training. Let us use a learning rate of 10^{-3} from now on.

TODO *(optional)*

- Run training of the same model used above with
  - batch size 1 for **one epoch**
  - batch size 512 for 100 epochs, using early stopping with patience 10
- Compare your training results of all three batch sizes you have trained, i.e. batch size 1, 512 and 1024 (from above)
- Was it smart to set batch size to 1?
- How long (in terms of computing time) do your models need to train for the different batch sizes? (You could even measure this with python, using the `time` package)
- What is the impact on model performance?

Hint: You have to initialize new data loaders, as they provide you with batches during training.

In [22]:
```
# Batch size 1

################################
## YOUR CODE HERE - OPTIONAL ##
################################
```

In [23]:
```
# Batch size 512

################################
## YOUR CODE HERE - OPTIONAL ##
################################
```

What about the architecture?

How does architecture affect predictive performance?

TODO:

In the following, try to improve model performance by varying

- number of hidden units
- number of layers
- activation function used

These parameters are called hyper-parameters, since they are excluded from model optimization. Instead, we have to set them by hand and explore them to find a model with good predictive accuracy.

Vary only one hyper-parameter at a time. If you would vary multiple parameters at the same time, it would be harder for you to see the impact that each parameter has.

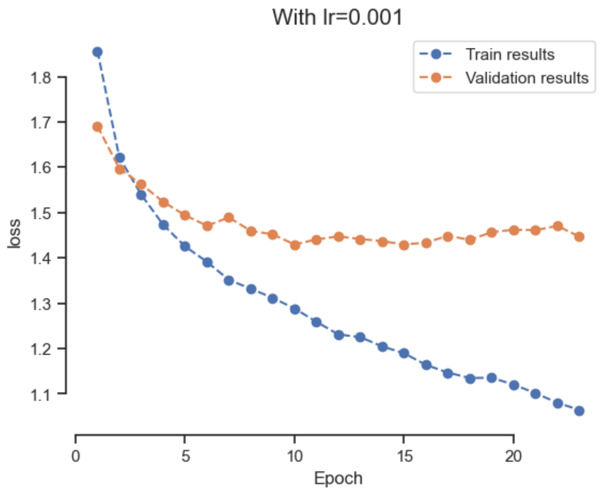
In [14]:

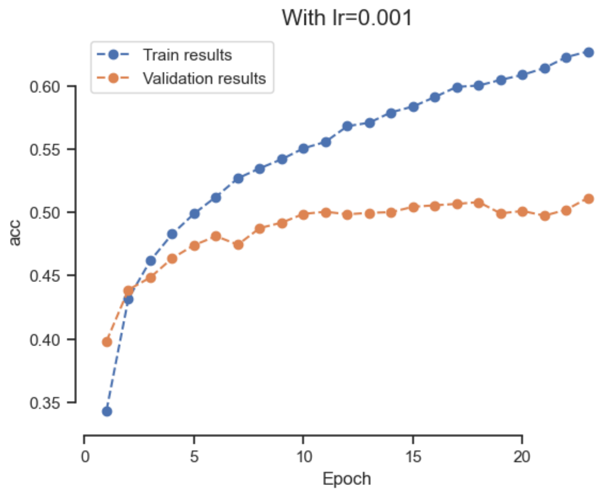
```
# number of hidden units

###################
lr = 1e-3
n_hidden_units = 60
patience = 4
loss_fn = torch.nn.CrossEntropyLoss()
save_path = 'saves/model_h_60.pt'
model = mlp(img_width=32, num_in_channels=3, num_classes=10, num_hidden_units=n_hidden_units, num_hidden_layers
optimizer = optim.Adam(model.parameters(), lr=lr)

train_losses_h_60, val_losses_h_60, train_accs_h_60, val_accs_h_60, _ = run_training(
  model=model,
  optimizer=optimizer,
  loss_function=loss_fn,
  device=device,
  num_epochs=100,
  train_dataloader=trainloader,
   val_dataloader=valloader,
   early_stopper=EarlyStopper(verbose=True, path=save_path, patience_number=patience)
)

plot(title=f'With lr={lr}', label='loss', train_results=train_losses_h_60, val_results=val_losses_h_60)
plot(title=f'With lr={lr}', label='acc', train_results=train_accs_h_60, val_results=val_accs_h_60)
plt.show()
###################
```

22.00% [22/100 04:10<14:49]

100.00% [10/10 00:03<00:00]

```
Checking acc: [0. 0. 0. 0.] < 0.0
Checking acc: [0.     0.     0.     0.3982] < 0.0
Checking acc: [0.     0.     0.3982 0.4388] < 0.0
Checking acc: [0.     0.3982 0.4388 0.4485] < 0.0
Checking acc: [0.3982 0.4388 0.4485 0.4637] < 0.0
Checking acc: [0.4388 0.4485 0.4637 0.474 ] < 0.3982
Checking acc: [0.4485 0.4637 0.474  0.4811] < 0.4388
Checking acc: [0.4637 0.474  0.4811 0.4744] < 0.4485
Checking acc: [0.474  0.4811 0.4744 0.4875] < 0.4637
Checking acc: [0.4811 0.4744 0.4875 0.4918] < 0.474
Checking acc: [0.4744 0.4875 0.4918 0.4987] < 0.4811
Checking acc: [0.4875 0.4918 0.4987 0.5004] < 0.4744
Checking acc: [0.4918 0.4987 0.5004 0.4984] < 0.4875
Checking acc: [0.4987 0.5004 0.4984 0.4995] < 0.4918
Checking acc: [0.5004 0.4984 0.4995 0.5001] < 0.4987
Checking acc: [0.4984 0.4995 0.5001 0.5043] < 0.5004
Checking acc: [0.4995 0.5001 0.5043 0.5055] < 0.4984
Checking acc: [0.5001 0.5043 0.5055 0.5066] < 0.4995
Checking acc: [0.5043 0.5055 0.5066 0.508 ] < 0.5001
Checking acc: [0.5055 0.5066 0.508  0.4991] < 0.5043
Checking acc: [0.5066 0.508  0.4991 0.501 ] < 0.5055
Checking acc: [0.508  0.4991 0.501  0.4973] < 0.5066
Checking acc: [0.4991 0.501  0.4973 0.502 ] < 0.508
```






```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [14], line 26
     24 plot(title=f'With lr={lr}', label='acc', train_results=train_accs_h_60, val_results=val_accs_h_60)
     25 plt.show()
---> 26 print_best(val_losses_lr_s, val_accs_lr_s)

NameError: name 'print_best' is not defined
```

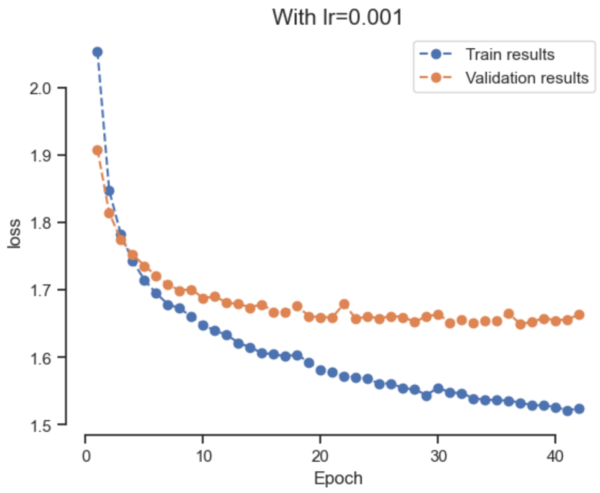
In [17]:
```
print_best(val_losses_h_60, val_accs_h_60)
lr = 1e-3
n_hidden_units = 10
patience = 4
loss_fn = torch.nn.CrossEntropyLoss()
save_path = 'saves/model_h_10.pt'
model = mlp(img_width=32, num_in_channels=3, num_classes=10, num_hidden_units=n_hidden_units, num_hidden_layers
optimizer = optim.Adam(model.parameters(), lr=lr)

train_losses_h_10, val_losses_h_10, train_accs_h_10, val_accs_h_10, _ = run_training(
  model=model,
  optimizer=optimizer,
  loss_function=loss_fn,
  device=device,
  num_epochs=100,
  train_dataloader=trainloader,
   val_dataloader=valloader,
   early_stopper=EarlyStopper(path=save_path, patience_number=patience)
)

plot(title=f'With lr={lr}', label='loss', train_results=train_losses_h_10, val_results=val_losses_h_10)
plot(title=f'With lr={lr}', label='acc', train_results=train_accs_h_10, val_results=val_accs_h_10)
plt.show()
print_best(val_losses_h_10, val_accs_h_10)
```

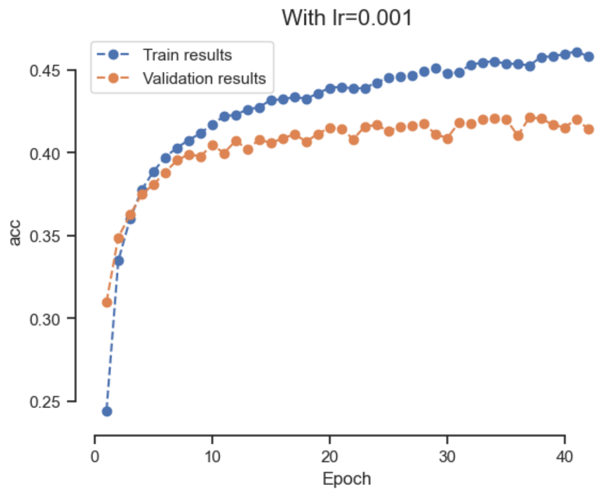
```
Minimal validation lost: 1.429413866996765, occured at epoch: 9
Maximal accuracy: 0.5114, occured at epoch: 22
```

41.00% [41/100 05:52<08:27]

100.00% [10/10 00:02<00:00]

```
Checking acc: [0. 0. 0. 0.] < 0.0
Checking acc: [0.   0.   0.   0.31] < 0.0
Checking acc: [0.    0.    0.31  0.349] < 0.0
Checking acc: [0.     0.31   0.349  0.3626] < 0.0
Checking acc: [0.31   0.349  0.3626 0.3749] < 0.0
Checking acc: [0.349  0.3626 0.3749 0.3809] < 0.31
Checking acc: [0.3626 0.3749 0.3809 0.3883] < 0.349
Checking acc: [0.3749 0.3809 0.3883 0.3955] < 0.3626
Checking acc: [0.3809 0.3883 0.3955 0.3991] < 0.3749
Checking acc: [0.3883 0.3955 0.3991 0.3979] < 0.3809
Checking acc: [0.3955 0.3991 0.3979 0.4048] < 0.3883
Checking acc: [0.3991 0.3979 0.4048 0.3999] < 0.3955
Checking acc: [0.3979 0.4048 0.3999 0.4073] < 0.3991
Checking acc: [0.4048 0.3999 0.4073 0.4022] < 0.3979
Checking acc: [0.3999 0.4073 0.4022 0.408 ] < 0.4048
Checking acc: [0.4073 0.4022 0.408  0.406 ] < 0.3999
Checking acc: [0.4022 0.408  0.406  0.4088] < 0.4073
Checking acc: [0.408  0.406  0.4088 0.4112] < 0.4022
Checking acc: [0.406  0.4088 0.4112 0.4068] < 0.408
Checking acc: [0.4088 0.4112 0.4068 0.4115] < 0.406
Checking acc: [0.4112 0.4068 0.4115 0.4152] < 0.4088
Checking acc: [0.4068 0.4115 0.4152 0.4147] < 0.4112
Checking acc: [0.4115 0.4152 0.4147 0.4082] < 0.4068
Checking acc: [0.4152 0.4147 0.4082 0.416 ] < 0.4115
Checking acc: [0.4147 0.4082 0.416  0.4168] < 0.4152
Checking acc: [0.4082 0.416  0.4168 0.4131] < 0.4147
Checking acc: [0.416  0.4168 0.4131 0.4158] < 0.4082
Checking acc: [0.4168 0.4131 0.4158 0.4166] < 0.416
Checking acc: [0.4131 0.4158 0.4166 0.4177] < 0.4168
Checking acc: [0.4158 0.4166 0.4177 0.4113] < 0.4131
Checking acc: [0.4166 0.4177 0.4113 0.4089] < 0.4158
Checking acc: [0.4177 0.4113 0.4089 0.4183] < 0.4166
Checking acc: [0.4113 0.4089 0.4183 0.4179] < 0.4177
Checking acc: [0.4089 0.4183 0.4179 0.42  ] < 0.4113
Checking acc: [0.4183 0.4179 0.42   0.421 ] < 0.4089
Checking acc: [0.4179 0.42   0.421  0.4205] < 0.4183
Checking acc: [0.42   0.421  0.4205 0.4104] < 0.4179
Checking acc: [0.421  0.4205 0.4104 0.4214] < 0.42
Checking acc: [0.4205 0.4104 0.4214 0.4207] < 0.421
Checking acc: [0.4104 0.4214 0.4207 0.4169] < 0.4205
Checking acc: [0.4214 0.4207 0.4169 0.4154] < 0.4104
Checking acc: [0.4207 0.4169 0.4154 0.4204] < 0.4214
```





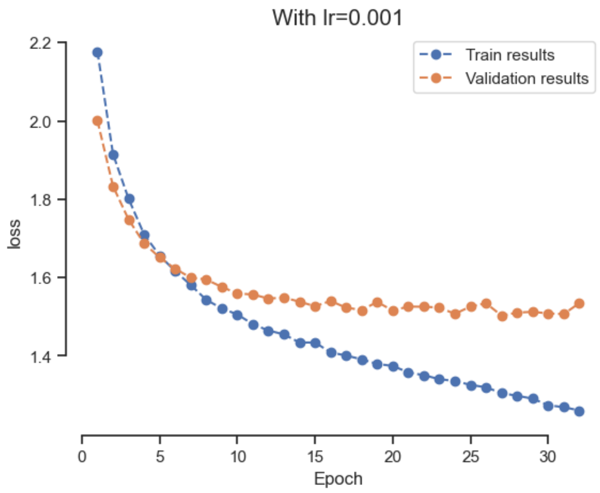

```
Minimal validation lost: 1.649216890335083, occured at epoch: 36
Maximal accuracy: 0.4214, occured at epoch: 36
```

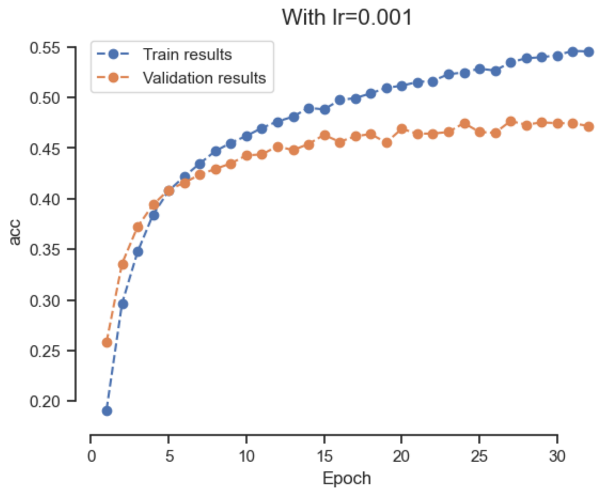
In [18]:

```
# number of layers

####################
lr = 1e-3
n_hidden_units = 30
n_layers = 5
patience = 4
loss_fn = torch.nn.CrossEntropyLoss()
save_path = 'saves/model_hl_5.pt'
model = mlp(img_width=32, num_in_channels=3, num_classes=10, num_hidden_units=n_hidden_units, num_hidden_layers
optimizer = optim.Adam(model.parameters(), lr=lr)

train_losses_hl_5, val_losses_hl_5, train_accs_hl_5, val_accs_hl_5, _ = run_training(
  model=model,
  optimizer=optimizer,
  loss_function=loss_fn,
  device=device,
  num_epochs=100,
  train_dataloader=trainloader,
   val_dataloader=valloader,
   early_stopper=EarlyStopper(path=save_path, patience_number=patience)
)

plot(title=f'With lr={lr}', label='loss', train_results=train_losses_hl_5, val_results=val_losses_hl_5)
plot(title=f'With lr={lr}', label='acc', train_results=train_accs_hl_5, val_results=val_accs_hl_5)
plt.show()
print_best(val_losses_hl_5, val_accs_hl_5)
####################
```

31.00% [31/100 04:28<09:57]

100.00% [10/10 00:03<00:00]





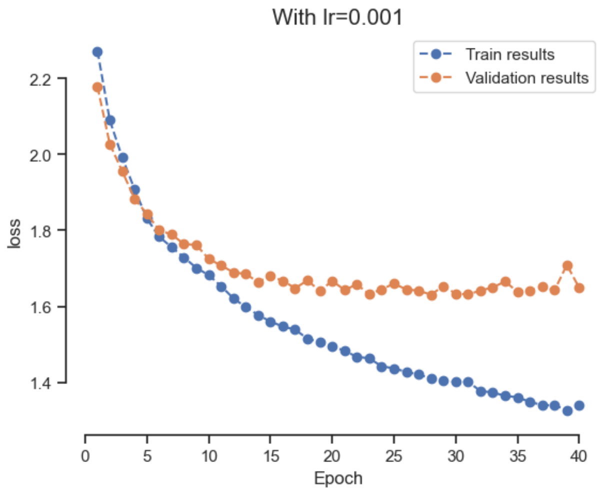

```
Minimal validation lost: 1.5028316617012023, occured at epoch: 26
Maximal accuracy: 0.4767, occured at epoch: 26
```

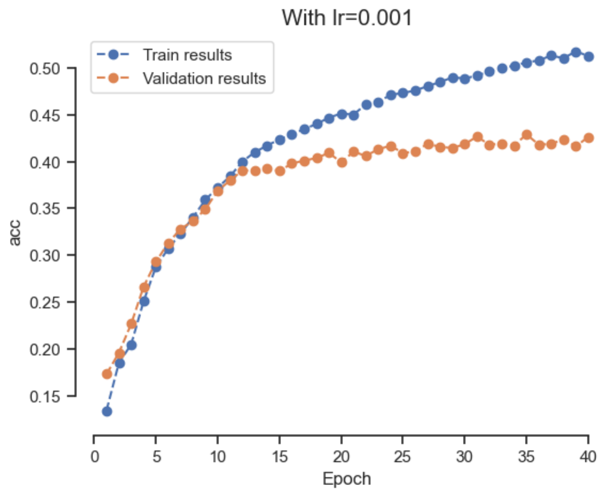
In [19]:

```
lr = 1e-3
n_hidden_units = 30
n_layers = 10
patience = 4
loss_fn = torch.nn.CrossEntropyLoss()
save_path = 'saves/model_hl_10.pt'
model = mlp(img_width=32, num_in_channels=3, num_classes=10, num_hidden_units=n_hidden_units, num_hidden_layers
optimizer = optim.Adam(model.parameters(), lr=lr)

train_losses_hl_10, val_losses_hl_10, train_accs_hl_10, val_accs_hl_10, _ = run_training(
  model=model,
  optimizer=optimizer,
  loss_function=loss_fn,
  device=device,
  num_epochs=100,
  train_dataloader=trainloader,
   val_dataloader=valloader,
   early_stopper=EarlyStopper(path=save_path, patience_number=patience)
)

plot(title=f'With lr={lr}', label='loss', train_results=train_losses_hl_10, val_results=val_losses_hl_10)
plot(title=f'With lr={lr}', label='acc', train_results=train_accs_hl_10, val_results=val_accs_hl_10)
plt.show()
print_best(val_losses_hl_10, val_accs_hl_10)
```

39.00% [39/100 05:47<09:03]

100.00% [10/10 00:03<00:00]





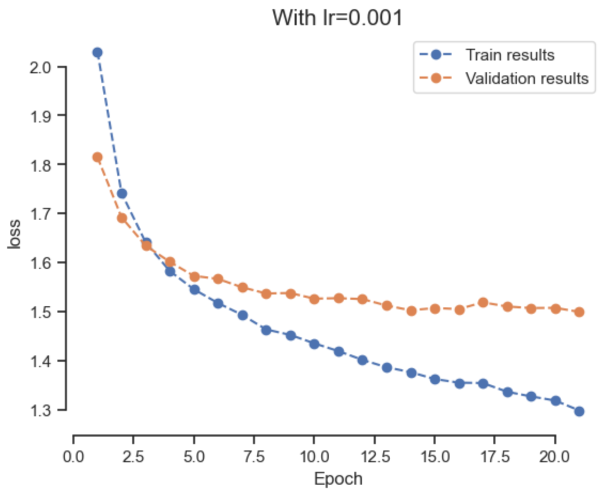

```
Minimal validation lost: 1.6308357119560242, occured at epoch: 27
Maximal accuracy: 0.429, occured at epoch: 34
```

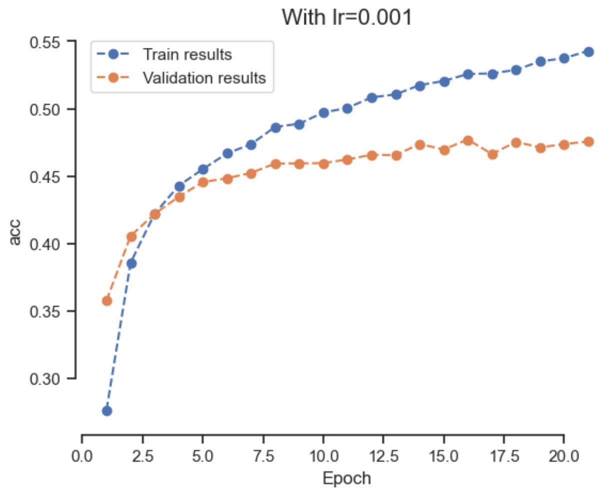
In [22]:

```
lr = 1e-3
n_hidden_units = 30
n_layers = 2
patience = 4
loss_fn = torch.nn.CrossEntropyLoss()
save_path = 'saves/model_hl_2.pt'
model = mlp(img_width=32, num_in_channels=3, num_classes=10, num_hidden_units=n_hidden_units, num_hidden_layers
optimizer = optim.Adam(model.parameters(), lr=lr)

train_losses_hl_2, val_losses_hl_2, train_accs_hl_2, val_accs_hl_2, _ = run_training(
  model=model,
  optimizer=optimizer,
  loss_function=loss_fn,
  device=device,
  num_epochs=100,
  train_dataloader=trainloader,
   val_dataloader=valloader,
   early_stopper=EarlyStopper(path=save_path, patience_number=patience)
)

plot(title=f'With lr={lr}', label='loss', train_results=train_losses_hl_2, val_results=val_losses_hl_2)
plot(title=f'With lr={lr}', label='acc', train_results=train_accs_hl_2, val_results=val_accs_hl_2)
plt.show()
print_best(val_losses_hl_2, val_accs_hl_2)
```

20.00% [20/100 02:49<11:18]

100.00% [10/10 00:02<00:00]





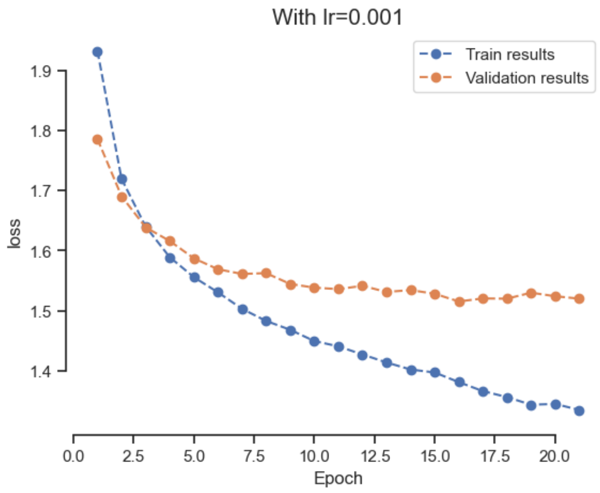

```
Minimal validation lost: 1.4999217987060547, occured at epoch: 20
Maximal accuracy: 0.477, occured at epoch: 15
```

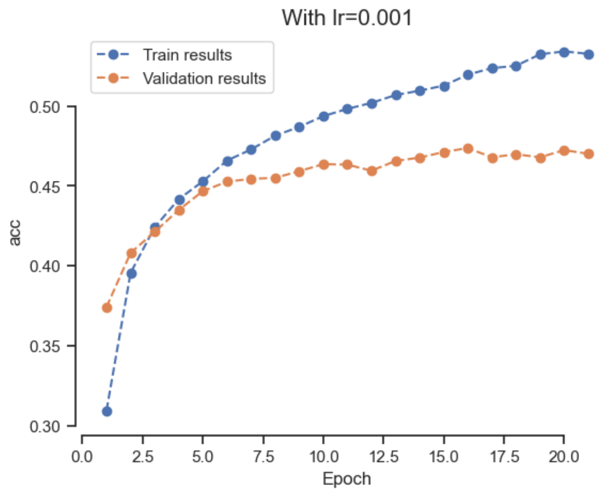
In [20]:

```
# activation function

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

lr = 1e-3
n_hidden_units = 30
n_layers = 1
patience = 4
loss_fn = torch.nn.CrossEntropyLoss()
save_path = 'saves/model_LRelu.pt'
model = mlp(img_width=32, num_in_channels=3, num_classes=10, num_hidden_units=n_hidden_units, num_hidden_layers
optimizer = optim.Adam(model.parameters(), lr=lr)

train_losses_LRelu, val_losses_LRelu, train_accs_LRelu, val_accs_LRelu, _ = run_training(
  model=model,
  optimizer=optimizer,
  loss_function=loss_fn,
  device=device,
  num_epochs=100,
  train_dataloader=trainloader,
   val_dataloader=valloader,
   early_stopper=EarlyStopper(path=save_path, patience_number=patience)
)

plot(title=f'With lr={lr}', label='loss', train_results=train_losses_LRelu, val_results=val_losses_LRelu)
plot(title=f'With lr={lr}', label='acc', train_results=train_accs_LRelu, val_results=val_accs_LRelu)
plt.show()
print_best(val_losses_LRelu, val_accs_LRelu)
####################
```

20.00% [20/100 02:52<11:29]

100.00% [10/10 00:03<00:00]





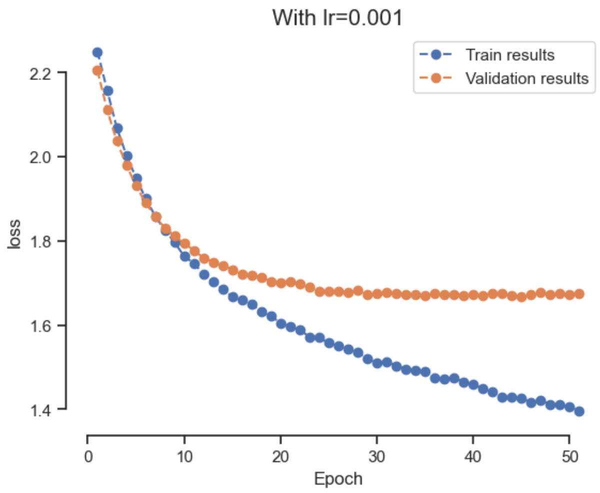

```
Minimal validation lost: 1.5151687264442444, occured at epoch: 15
Maximal accuracy: 0.4738, occured at epoch: 15
```

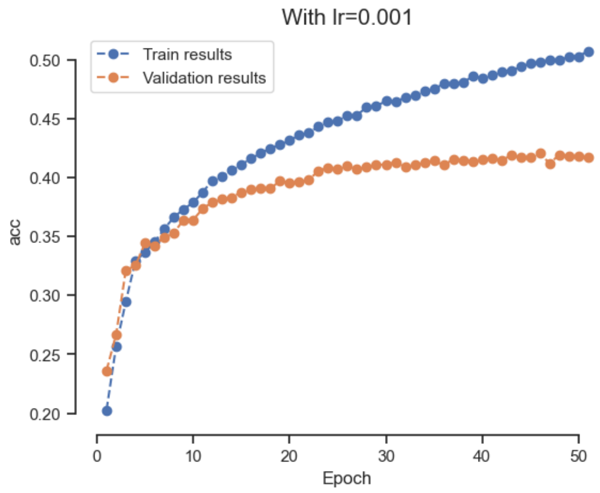
In [21]:

```
lr = 1e-3
n_hidden_units = 30
n_layers = 1
patience = 4
loss_fn = torch.nn.CrossEntropyLoss()
save_path = 'saves/model_sigmoid.pt'
model = mlp(img_width=32, num_in_channels=3, num_classes=10, num_hidden_units=n_hidden_units, num_hidden_layers
optimizer = optim.Adam(model.parameters(), lr=lr)

train_losses_sigmoid, val_losses_sigmoid, train_accs_sigmoid, val_accs_sigmoid, _ = run_training(
  model=model,
  optimizer=optimizer,
  loss_function=loss_fn,
  device=device,
  num_epochs=100,
  train_dataloader=trainloader,
   val_dataloader=valloader,
   early_stopper=EarlyStopper(path=save_path, patience_number=patience)
)

plot(title=f'With lr={lr}', label='loss', train_results=train_losses_sigmoid, val_results=val_losses_sigmoid)
plot(title=f'With lr={lr}', label='acc', train_results=train_accs_sigmoid, val_results=val_accs_sigmoid)
plt.show()
print_best(val_losses_sigmoid, val_accs_sigmoid)
```

50.00% [50/100 07:00<07:00]

100.00% [10/10 00:02<00:00]






```
Minimal validation lost: 1.6667879700660706, occured at epoch: 44
Maximal accuracy: 0.4207, occured at epoch: 45
```

Questions

- How good do you get?
- Which hyper-parameter makes the largest difference?
- Does it always help to make your model bigger (i.e. wider / deeper)? Why not?

Your answers:

- ...

Now, here are more TODO's, questions and a little challenge for you:

TODO

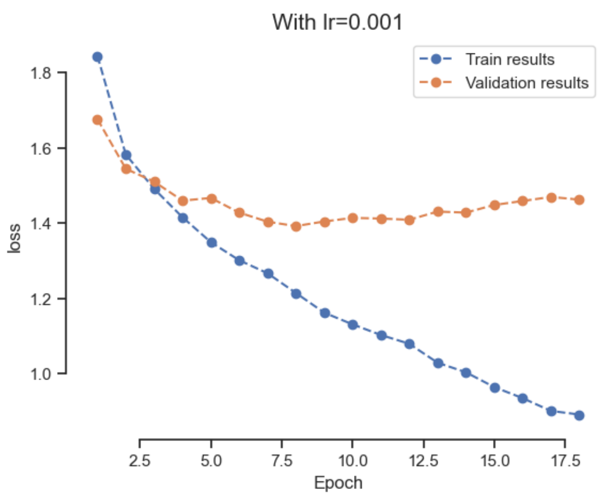
- If you choose your best values for number hidden units, number of layers and activation function that you determined by varying them independently above: Does performance improve? Why?
- Vary all of the parameters at the same time to maximize the predictive performance of your model. How good do you get?
  - When creating the exercise, I got a validation accuracy of 57%
  - Surpassing 50% val. acc. should be possible for you

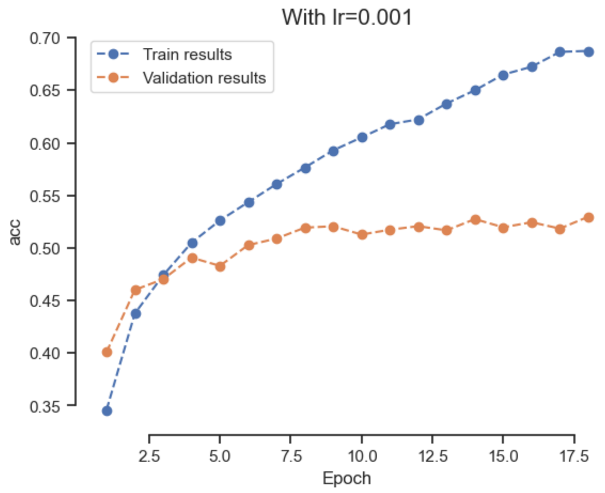
In [25]:

```
# Your best model:

####################
lr = 1e-3
n_hidden_units = 128
n_layers = 2
patience = 4
loss_fn = torch.nn.CrossEntropyLoss()
save_path = 'saves/model_best.pt'
model = mlp(img_width=32, num_in_channels=3, num_classes=10, num_hidden_units=n_hidden_units, num_hidden_layers
optimizer = optim.Adam(model.parameters(), lr=lr)

train_losses_best, val_losses_best, train_accs_best, val_accs_best, _ = run_training(
  model=model,
  optimizer=optimizer,
  loss_function=loss_fn,
  device=device,
  num_epochs=100,
  train_dataloader=trainloader,
   val_dataloader=valloader,
   early_stopper=EarlyStopper(path=save_path, patience_number=patience)
)

plot(title=f'With lr={lr}', label='loss', train_results=train_losses_best, val_results=val_losses_best)
plot(title=f'With lr={lr}', label='acc', train_results=train_accs_best, val_results=val_accs_best)
plt.show()
print_best(val_losses_best, val_accs_best)
####################
```

17.00% [17/100 02:25<11:48]

100.00% [10/10 00:02<00:00]





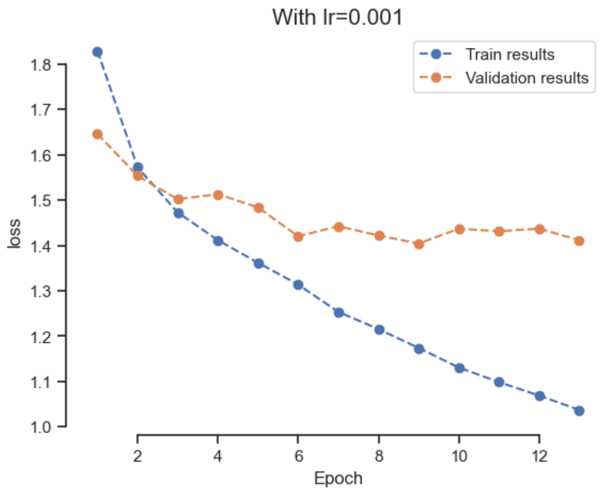

```
Minimal validation lost: 1.3919784665107726, occured at epoch: 7
Maximal accuracy: 0.5296, occured at epoch: 17
```

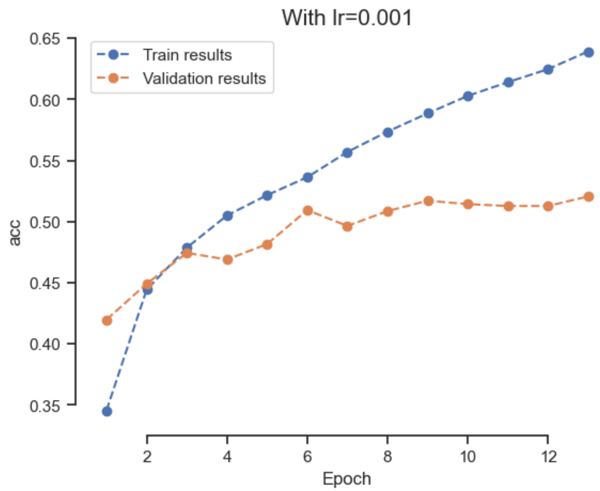
In [28]:

```
lr = 1e-3
n_hidden_units = 128
n_layers = 2
patience = 4
loss_fn = torch.nn.CrossEntropyLoss()
save_path = 'saves/model_best.pt'
model = mlp(img_width=32, num_in_channels=3, num_classes=10, num_hidden_units=n_hidden_units, num_hidden_layers
optimizer = optim.Adam(model.parameters(), lr=lr)

train_losses_best, val_losses_best, train_accs_best, val_accs_best, _ = run_training(
  model=model,
  optimizer=optimizer,
  loss_function=loss_fn,
  device=device,
  num_epochs=100,
  train_dataloader=trainloader,
   val_dataloader=valloader,
   early_stopper=EarlyStopper(path=save_path, patience_number=patience)
)

plot(title=f'With lr={lr}', label='loss', train_results=train_losses_best, val_results=val_losses_best)
plot(title=f'With lr={lr}', label='acc', train_results=train_accs_best, val_results=val_accs_best)
plt.show()
print_best(val_losses_best, val_accs_best)
```

12.00% [12/100 01:43<12:40]

100.00% [10/10 00:03<00:00]






```
Minimal validation lost: 1.4044419527053833, occured at epoch: 8
Maximal accuracy: 0.5206, occured at epoch: 12
```

Questions:

- If you train the same model multiple times from scratch: do you get the same performance? Are the models you trained above comparable then?
- What could we do about this?
  - *Hint:* there are actually multiple answers to this question.
  - One could be to change model training and evaluation. How?
  - The other could be to use a more sophisticated analysis. How?

Evaluate your best model on test set, once!

When doing a study, at the very end right before writing up your paper, you evaluate the best model you chose on the test set. This is the performance value you will report to the public.

TODO

- What is the accuracy of the best model you found on the test set?
- Plot the confusion matrix, too! *(optional)*

In [29]:

```
def test(test_loader, model, device):
    """Compute accuracy and confusion matrix on test set.

    Args:
        test_loader (DataLoader): torch DataLoader of test set
        model (nn.Module): Model to evaluate on test set
        device (torch.device): Device to use

    Returns:
        float, torch.Tensor shape (10,10): Returns model accuracy on test set
            (percent classified correctly) and confusion matrix
    """

    ####################
    epoch_loss = []
    epoch_correct, epoch_total = 0, 0
    confusion_matrix = torch.zeros(10, 10)

    model.eval()
    with torch.no_grad():
        for x, y in fastprogress.progress_bar(test_loader, parent=None):
            # make a prediction on validation set
            y_pred = model(x.to(device))

            # For calculating the accuracy, save the number of correctly
            # classified images and the total number
            epoch_correct += sum(y.to(device) == y_pred.argmax(dim=1))
            epoch_total += len(y)

            # Fill confusion matrix
            for (y_true, y_p) in zip(y, y_pred.argmax(dim=1)):
                confusion_matrix[int(y_true), int(y_p)] +=1

            # Compute loss
            loss = loss_fn(y_pred, y.to(device))

            # For plotting the train loss, save it for each sample
            epoch_loss.append(loss.item())

    # Return the mean loss, the accuracy and the confusion matrix
    return np.mean(epoch_loss), accuracy(epoch_correct, epoch_total), confusion_matrix
    ####################
```

In [30]:

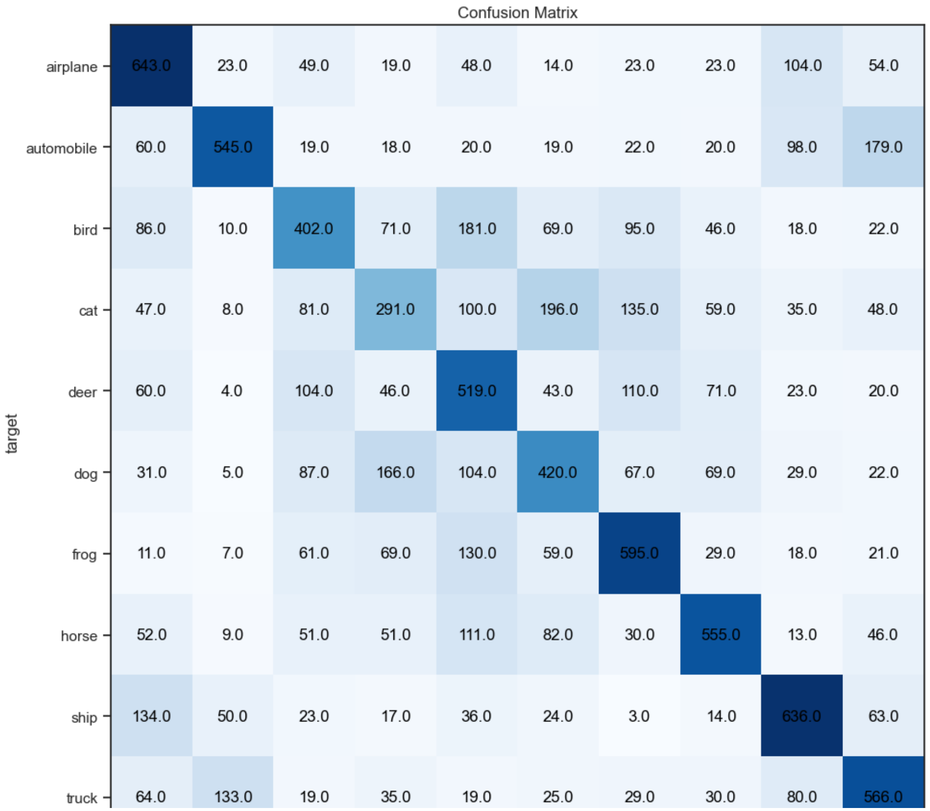
```
# The accuracy and confusion matrix

####################
test_loss, test_acc, test_confusuion = test(testloader, model, device)
print(f"Test loss: {test_loss} and accuracy: {test_acc}")
####################
```

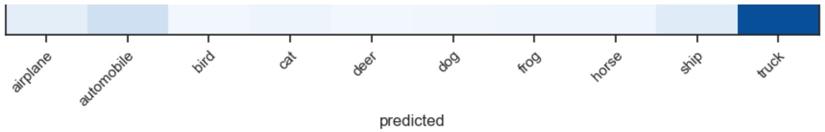
100.00% [10/10 00:03<00:00]

```
Test loss: 1.4096778154373169 and accuracy: 0.5172
```

In [35]:

```
fig, ax = plt.subplots(figsize=(10,10))
im = ax.imshow(test_confusuion, cmap='Blues')

# Show all ticks and label them with the respective list entries

ax.set_xticks(np.arange(10))
ax.set_yticks(np.arange(10))

ax.set_xticklabels(cifar10_train.classes)
ax.set_yticklabels(cifar10_train.classes)

ax.set_xlabel('predicted')
ax.set_ylabel('target')

# Rotate the tick labels and set their alignment.
plt.setp(ax.get_xticklabels(), rotation=45, ha="right",
         rotation_mode="anchor")

# Loop over data dimensions and create text annotations.
for i in range(10):
    for j in range(10):
        text = ax.text(j, i, float(test_confusuion[i, j]),
                       ha="center", va="center", color="black")

ax.set_title("Confusion Matrix")
fig.tight_layout()
plt.show()
```





Questions:

- On which classes is your model's prediction poor?
- Is the test accuracy of your model as good as the validation accuracy?
- If those values are different: How can you explain the difference?
- Why should you never use test set performance when trying out different hyper-parameters and architectures?