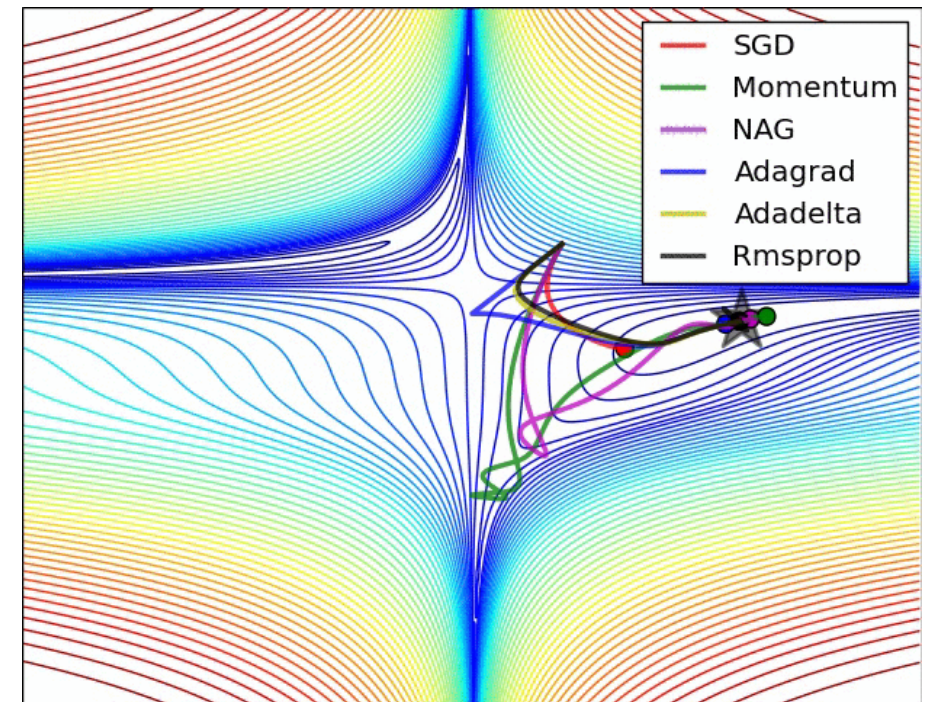


The background of the slide is a dark navy blue, filled with a bokeh effect of out-of-focus light circles. These circles vary in size and are primarily a warm yellow-gold color, with some cooler blue and teal tones interspersed, particularly towards the top and right edges. The overall effect is a soft, textured, and modern aesthetic.

Training Neural Networks

Contents

- Data Exploration
- Debugging the pipeline
- Train-validation-test split
- Overfitting
- Faster training



Data Exploration

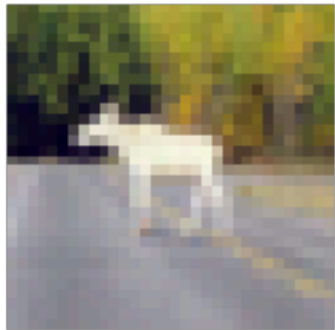
Data Exploration

To train a Neural Network in a supervised way we need data, and it is a good idea to explore it before anything else. In the case of images, we are interested in:

- Number of classes
- Number of channels (gray, RGB, multispectral, ...)
- Resolution (height, width)
- Statistics (mean/std, max/min values)
- Data type (uint8, float16, ...)
- Class distribution
- Any other aspect that may be interesting for your particular case.

CIFAR10

deer



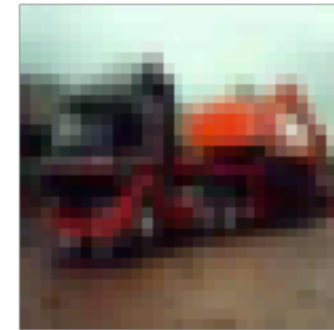
car



frog



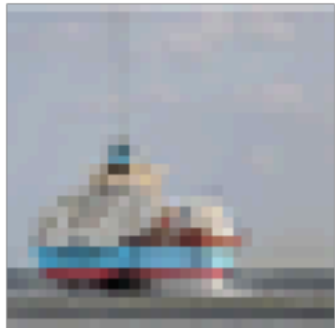
truck



deer



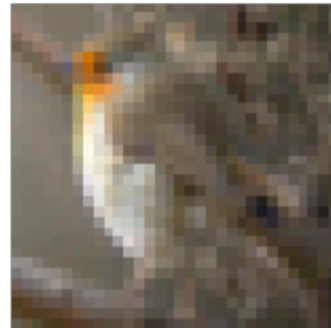
ship



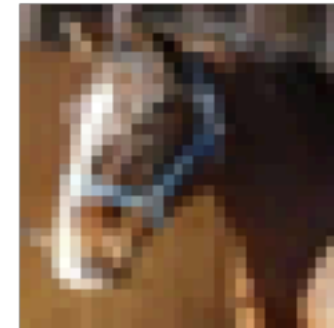
frog



bird



horse



cat



deer



ship



dog



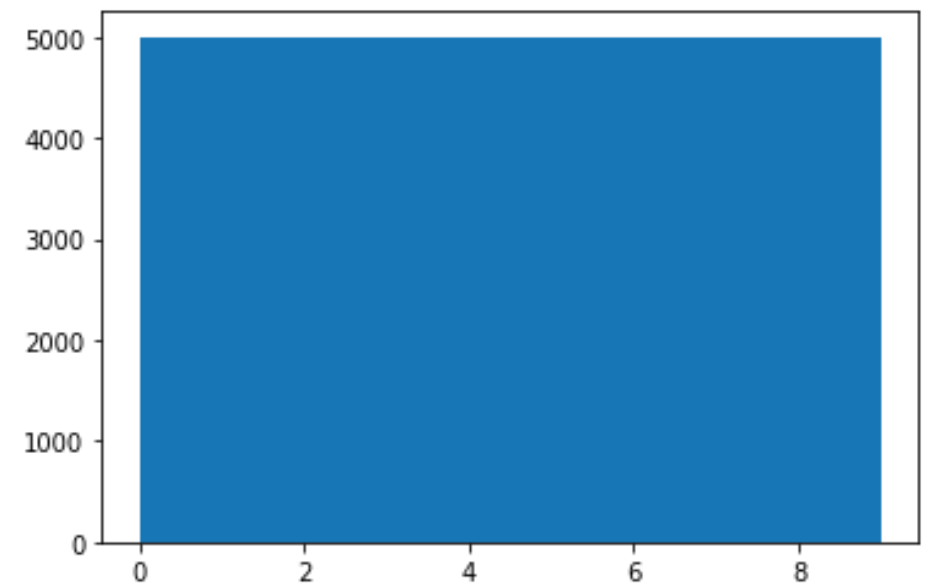
plane



ship



- Train/Test samples: 50000/10000
- Classes (10): plane, car, bird, cat, deer, dog, frog, horse, ship, truck
- Image shape: 32 x 32 x 3 pixels (RGB)
- Data Type: uint8 (integer values between 0-255)
- Stats: mean/std -> [0.49139968, 0.48215841, 0.44653091], [0.24703223, 0.24348513, 0.26158784]
- Class distribution: Balanced dataset (5000 samples per class)



Debugging the pipeline

Debugging the pipeline

We want to start small and improve sequentially to avoid bugs (some bugs can be very difficult to spot because the network will always learn something as long as dimensions match).

- Fit only one sample
- Fit only one batch of samples
- Experiment with a representative subset
- Only then, train on the entire dataset

Fit one sample

This step will spot early on problems such as:

- Bad definition of the Dataset or DataLoader.
- Miss-shaped tensor dimensions on the data or the network.
- Wrong loss function

```
class Dataset(torch.utils.data.Dataset):
    def __init__(self, X, stats, y=None, train=True):
        self.X = X
        self.mean, self.std = stats
        self.y = y
        self.train = train

    def __len__(self):
        return len(self.X)

    def __getitem__(self, ix):
        img = self.X[ix]
        img = torch.from_numpy(img / 255.)
        img = (img - self.mean) / self.std
        img = img.view(-1).float()
        if self.train:
            label = torch.tensor(self.y[ix]).long()
            return img, label
        return img
```

```
train_dataset = Dataset(train_images[:1], (mean, std), train_labels[:1])
```

```
len(train_dataset)
```

```
> 1
```

- Check the outputs of the Dataset to make sure dimensions are correct.
- Compare the image given by the Dataset with the original.
- Check the outputs of the DataLoader to make sure dimensions are correct.

```
img, label = train_dataset[0]

img.shape, label.shape

> torch.Size([3072]), torch.Size([])

# retrieve original image

img = img.reshape(32, 32, 3)
img = img*std + mean
```

```
train_dataloader = DataLoader(train_dataset, shuffle=True, batch_size=3)
```

```
batch = next(iter(train_dataloader))
imgs, labels = batch
```

```
imgs.shape, labels.shape
```

```
> torch.Size([1, 3072]), torch.Size([1])
```

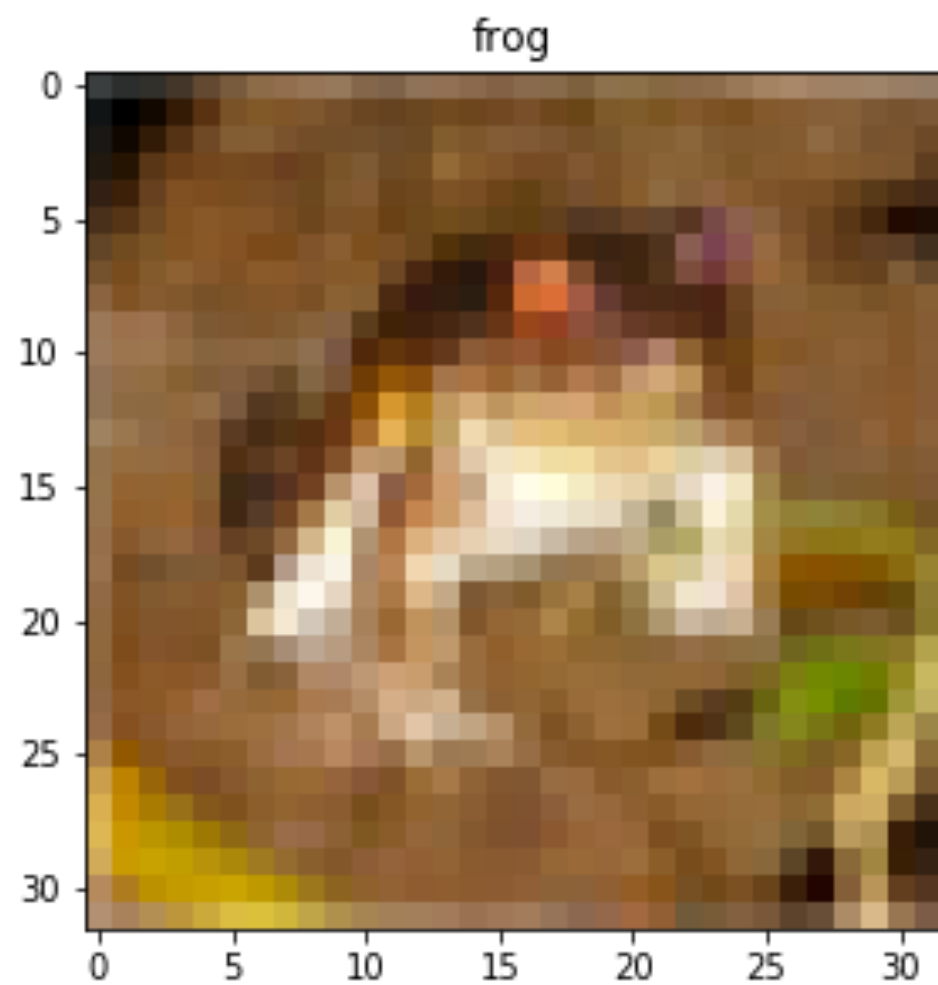
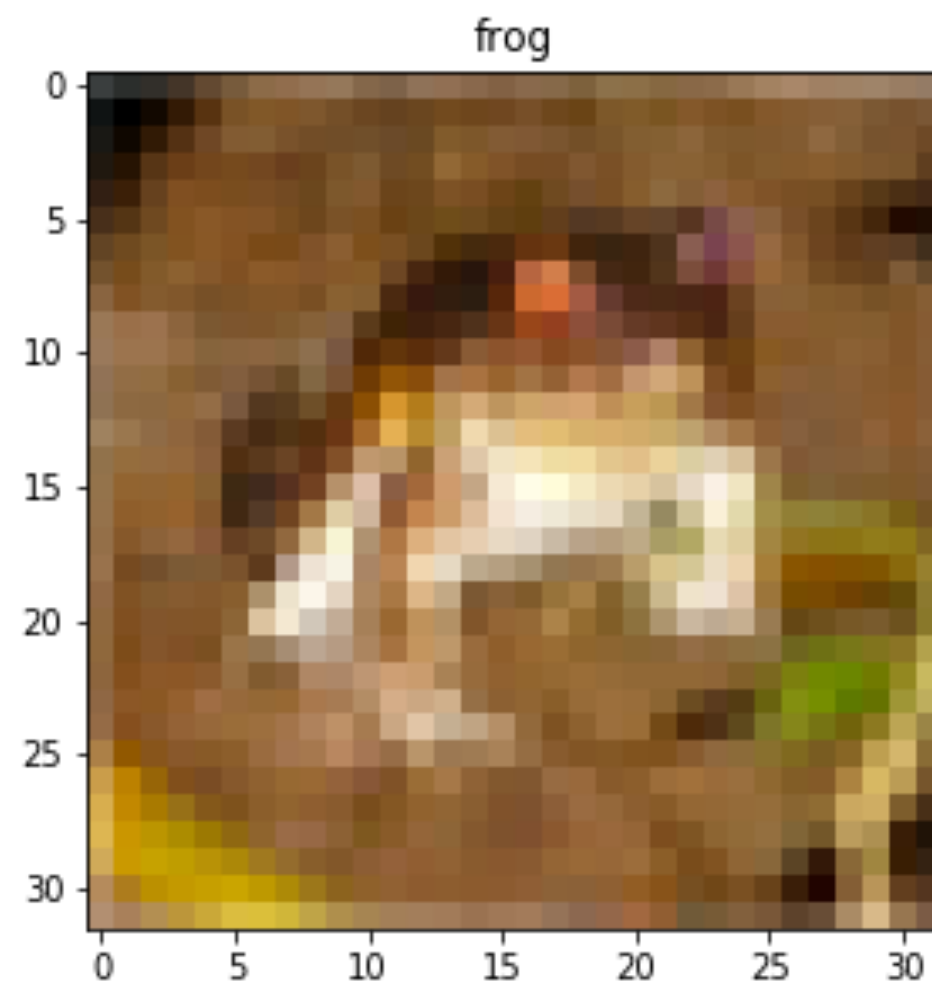


Image given by dataset



Original image

Train a network and make sure you it can fit just one image.
If not, something is wrong (check dimensions, loss function,
training loop, ...) !

```
net = torch.nn.Sequential(  
    torch.nn.Linear(32*32*3, 100),  
    torch.nn.ReLU(),  
    torch.nn.Linear(100, 10)  
)  
  
model = Model(net)  
  
model.compile(optimizer = torch.optim.SGD(model.net.parameters(), lr=0.01),  
              loss = torch.nn.CrossEntropyLoss(),  
              metrics=[Accuracy()])  
  
hist = model.fit(train_dataloader, epochs=3)  
  
> Epoch 1/3 loss 2.37651 acc 0.0  
> Epoch 2/3 loss 0.83177 acc 1.0  
> Epoch 3/3 loss 0.27745 acc 1.0
```

Fit one batch

```
batch_size = 16

train_dataset = Dataset(train_images[:batch_size], (mean, std), train_labels[:batch_size])

train_dataloader = DataLoader(train_dataset, shuffle=True, batch_size=batch_size)

batch = next(iter(train_dataloader))
imgs, labels = batch

len(train_dataset), imgs.shape, labels.shape
> 16, torch.Size([16, 3072]), torch.Size([16])

# define and compile a new model
# . . .

hist = model.fit(train_dataloader, epochs=5)

> Epoch 1/5 loss 2.39299 acc 0.0625
> Epoch 2/5 loss 2.11566 acc 0.3125
> Epoch 3/5 loss 1.89437 acc 0.6875
> Epoch 4/5 loss 1.71263 acc 0.875
> Epoch 5/5 loss 1.5552 acc 1.0
```

frog / frog



truck / truck



truck / truck



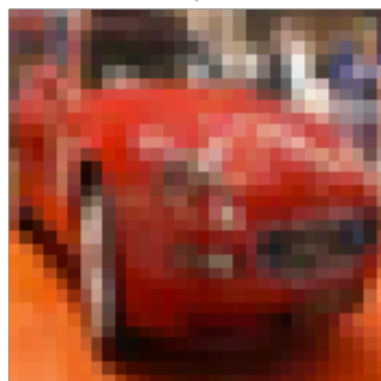
deer / deer



car / car



car / car



bird / bird



horse / horse



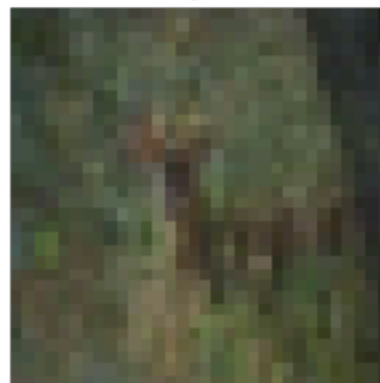
ship / ship



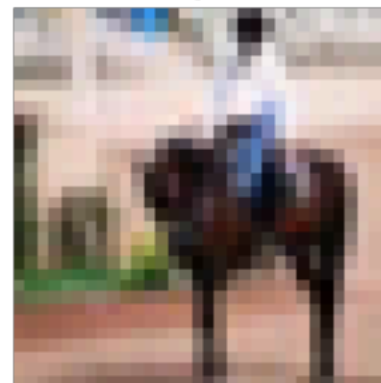
cat / cat



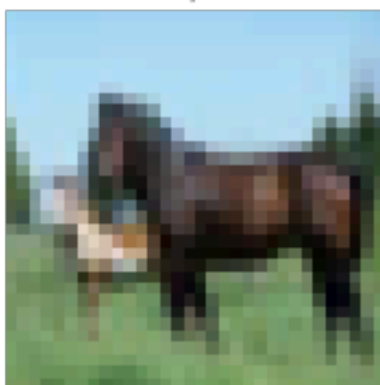
deer / deer



horse / horse



horse / horse



bird / bird



truck / truck



truck / truck



Representative subset

- We want a small representative subset so we can experiment quickly try different networks, optimizers, batch sizes, hyperparameters, etc.
- The idea is that the observed behavior should translate to the entire dataset.
- Keep the same class distribution, and get enough data to iterate over a bunch of batches.

```
from sklearn.model_selection import train_test_split

_, subset_images, _, subset_labels = train_test_split(train_images, train_labels,
test_size=0.1, shuffle=True, stratify=train_labels, random_state=42)

subset_images.shape, subset_labels.shape

> (5000, 32, 32, 3), (5000,)
```



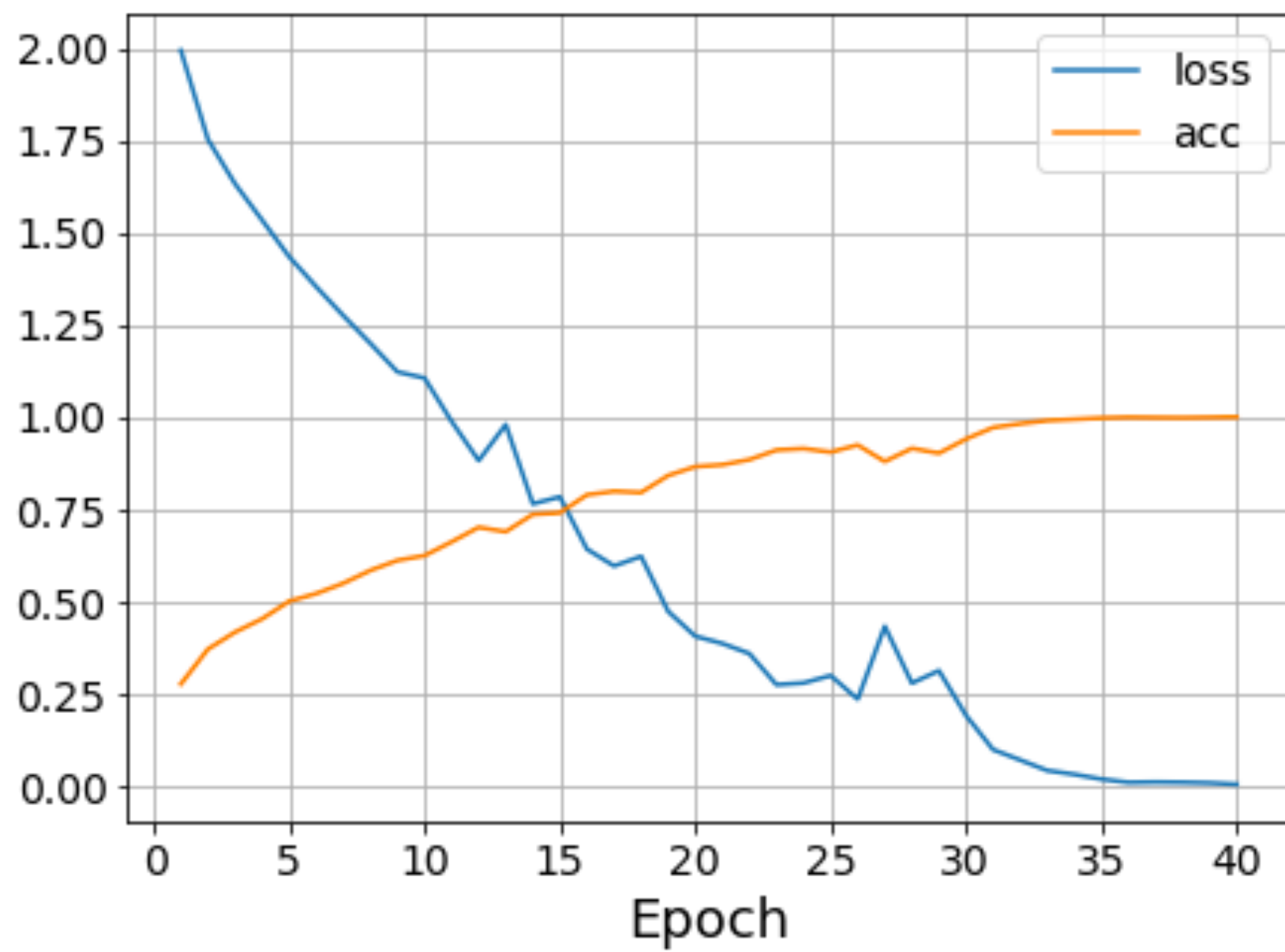
```
net = torch.nn.Sequential(  
    torch.nn.Linear(32*32*3, 100),  
    torch.nn.ReLU(),  
    torch.nn.Linear(100, 100),  
    torch.nn.ReLU(),  
    torch.nn.Linear(100, 10)  
)
```

```
model = Model(net)
```

```
model.compile(optimizer = torch.optim.SGD(model.net.parameters(), lr=0.1),  
              loss = torch.nn.CrossEntropyLoss(),  
              metrics=[Accuracy()])
```

```
hist = model.fit(train_dataloader, epochs=40)
```

```
> Epoch 1/40 loss 1.9933 acc 0.2767  
> Epoch 2/40 loss 1.75215 acc 0.37124  
> Epoch 3/40 loss 1.63208 acc 0.41733  
> . . .  
> Epoch 38/40 loss 0.0104 acc 0.99822  
> Epoch 39/40 loss 0.00885 acc 0.99901  
> Epoch 40/40 loss 0.00508 acc 1.0
```



WOW ! 100% accuracy ! Did we train the best classifier ever ?

```
test_dataset = Dataset(test_images, (mean, std), test_labels)
test_dataloader = DataLoader(test_dataset, shuffle=False, batch_size=batch_size)

model.evaluate(test_dataloader)

> eval_loss 4.6196 eval_acc 0.40625
```


The model works very well with **training** data, but very bad on the **testing** data... we need to evaluate the model during training on unseen data (not used for training). We need **validation** data.

Train-validation-test data splitting

We extract our validation data from the training set. It should be representative (same class distribution, between 10-30% is usually used).

```
train_images, eval_images, train_labels, eval_labels =  
train_test_split(train_images, train_labels, test_size=0.2, shuffle=True,  
stratify=train_labels, random_state=42)
```

```
train_images.shape, eval_images.shape, train_labels.shape, eval_labels.shape  
> (40000, 32, 32, 3), (10000, 32, 32, 3), (40000,), (10000,)
```

 Remember to get a new representative subset for fast experimentation from the new training set.

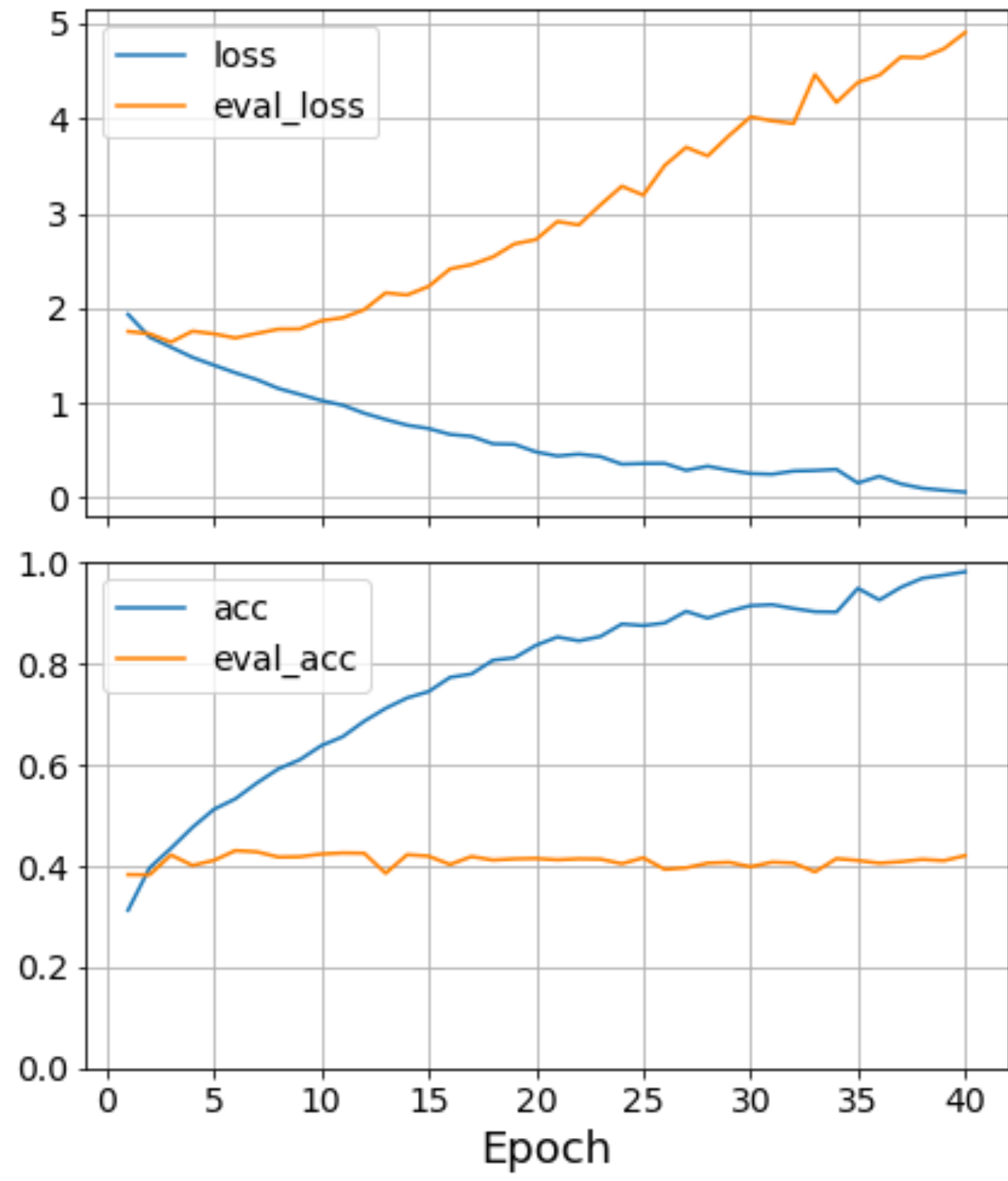
```
net = torch.nn.Sequential(
    torch.nn.Linear(32*32*3, 100),
    torch.nn.ReLU(),
    torch.nn.Linear(100, 100),
    torch.nn.ReLU(),
    torch.nn.Linear(100, 10)
)
```

```
model = Model(net)
```

```
model.compile(optimizer = torch.optim.SGD(model.net.parameters(), lr=0.1),
              loss = torch.nn.CrossEntropyLoss(),
              metrics=[Accuracy()])
```

```
hist = model.fit(train_dataloader, eval_dataloader, epochs=40)
```

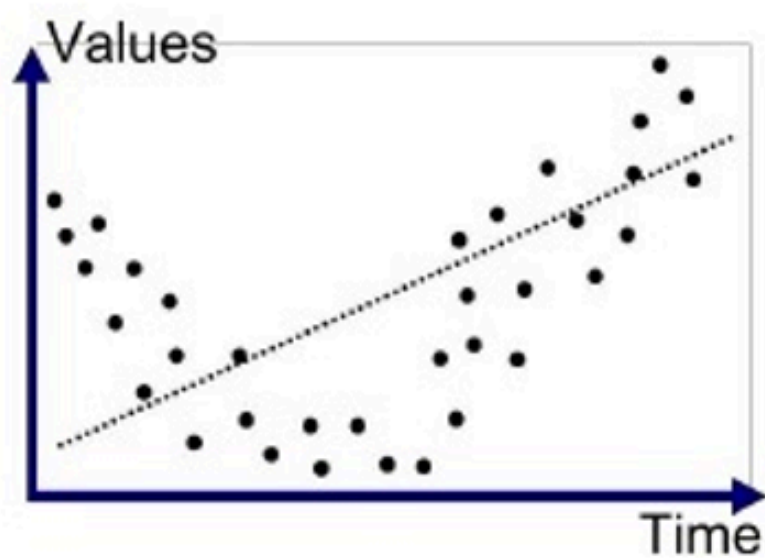
```
> Epoch 1/40 loss 1.93342 acc 0.31262 eval_loss 1.75308 eval_acc 0.3831
> Epoch 2/40 loss 1.69381 acc 0.39625 eval_loss 1.72813 eval_acc 0.3827
> Epoch 3/40 loss 1.58733 acc 0.43525 eval_loss 1.64047 eval_acc 0.422
> . . .
> Epoch 38/40 loss 0.09994 acc 0.9695 eval_loss 4.63953 eval_acc 0.4132
> Epoch 39/40 loss 0.0782 acc 0.9755 eval_loss 4.73261 eval_acc 0.4107
> Epoch 40/40 loss 0.06012 acc 0.98225 eval_loss 4.90485 eval_acc 0.4207
```



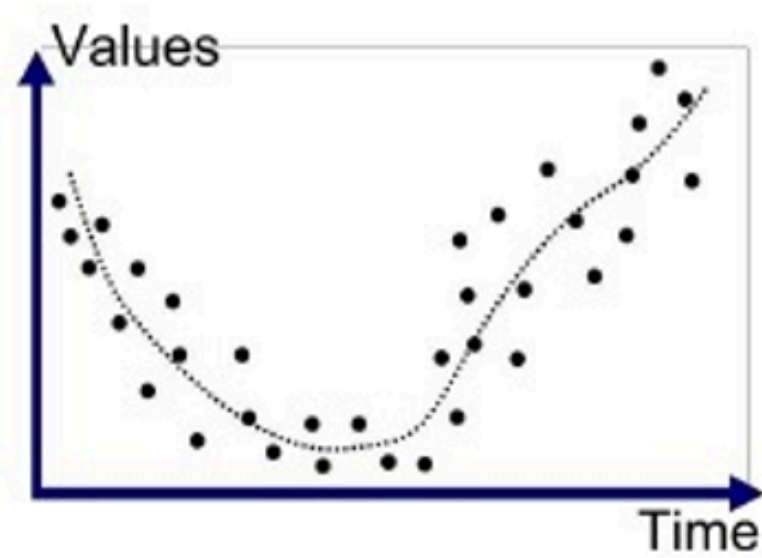
Overfitting

Overfitting

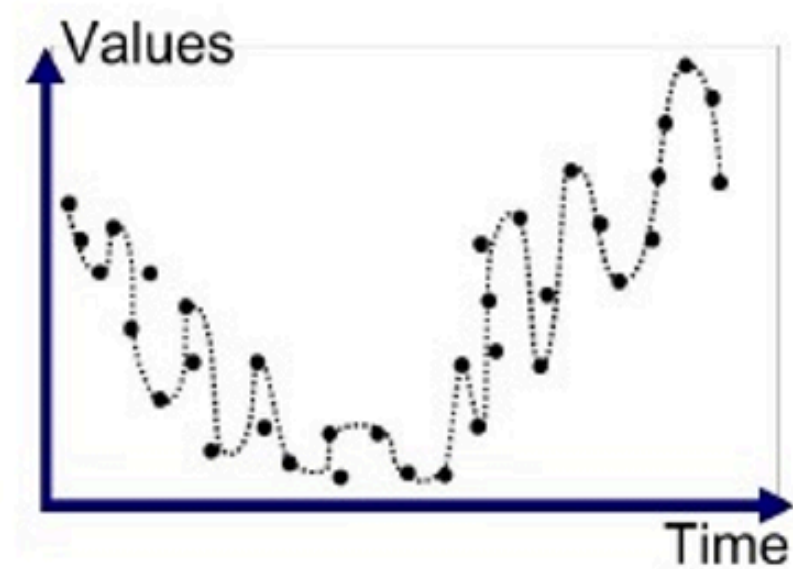
- Overfitting is the main problem that we have when training neural networks.
- This problem arises when the model learns the training data so well that loses generalization (good performance on unseen data).
- We can observe it from the [learning curves](#), the training metrics will improve while the validation metrics will not.



Underfitted



Good Fit/Robust



Overfitted

Reducing overfitting

- L2 Regularization
- Early stopping
- Dropout
- Use more data
- Data augmentation

L2 Regularization

- Regularization refer to any technique that reduces overfitting.
- L2 regularization constrains the magnitude of the parameters through a regularization term on the loss function.

$$l = CE(\hat{y}, y) + \alpha \frac{1}{2} ||\mathbf{w}||_2$$

```
net = torch.nn.Sequential(
    torch.nn.Linear(32*32*3, 100),
    torch.nn.ReLU(),
    torch.nn.Linear(100, 100),
    torch.nn.ReLU(),
    torch.nn.Linear(100, 10)
)

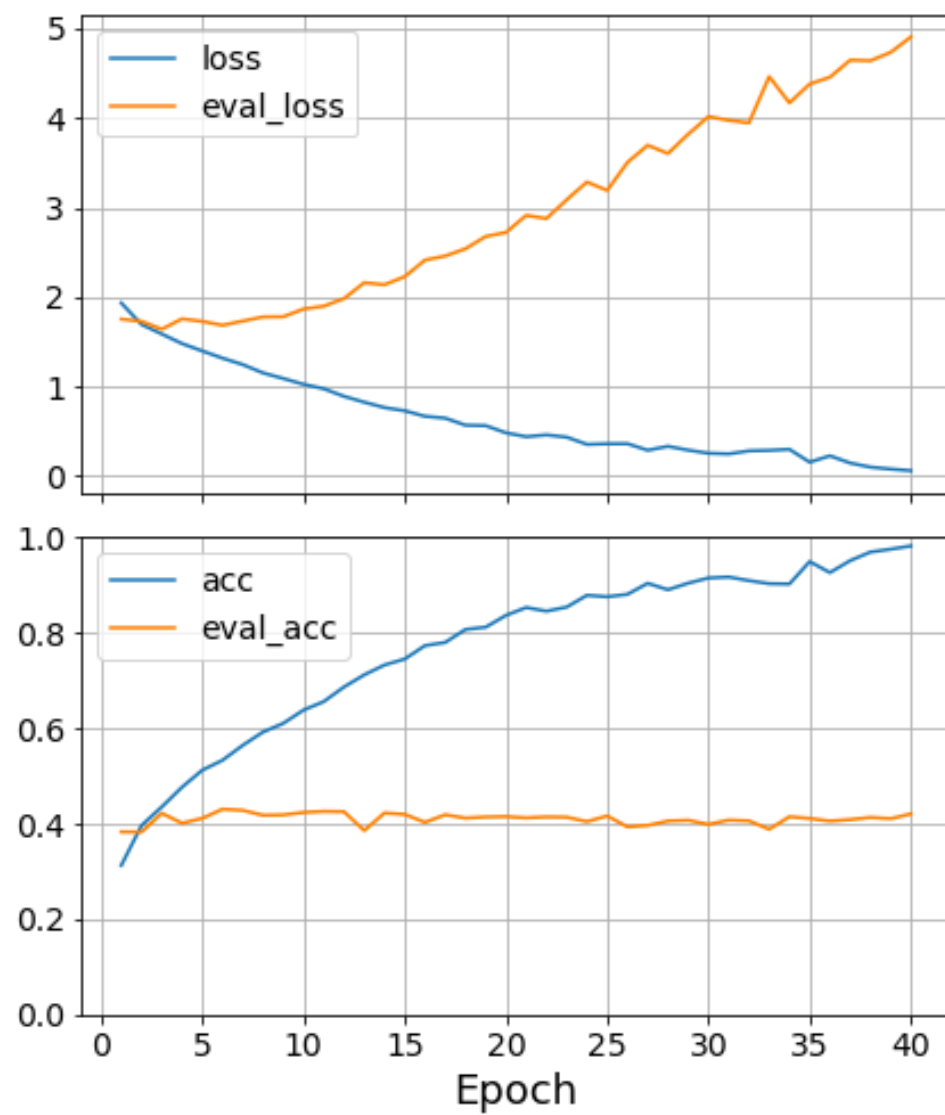
model = Model(net)

model.compile(optimizer = torch.optim.SGD(model.net.parameters(), lr=0.1, weight_decay=0.01),
              loss = torch.nn.CrossEntropyLoss(),
              metrics=[Accuracy()])

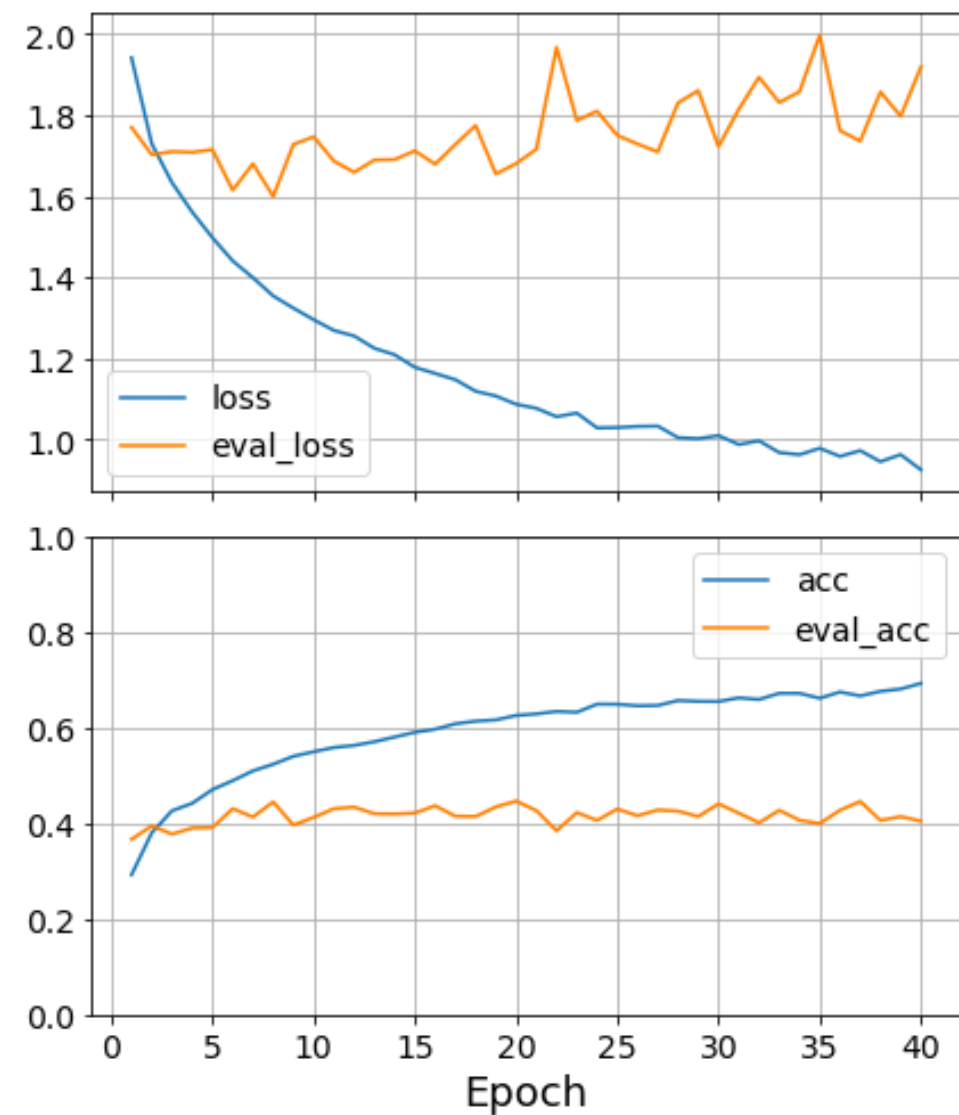
hist = model.fit(train_data_loader, eval_data_loader, epochs=40)

> Epoch 1/40 loss 1.9422 acc 0.29238 eval_loss 1.77027 eval_acc 0.3667
> Epoch 2/40 loss 1.72995 acc 0.38038 eval_loss 1.70297 eval_acc 0.3948
> Epoch 3/40 loss 1.63388 acc 0.4265 eval_loss 1.71049 eval_acc 0.3776
> . . .
> Epoch 38/40 loss 0.94518 acc 0.67638 eval_loss 1.85767 eval_acc 0.4065
> Epoch 39/40 loss 0.96268 acc 0.68138 eval_loss 1.79765 eval_acc 0.4143
> Epoch 40/40 loss 0.92502 acc 0.693 eval_loss 1.91941 eval_acc 0.405
```

No Weight Decay



Weight Decay



Early stopping

- Early stopping consists on keeping track of a metric during training (normally the validation loss) and stop the training loop when a determined number of epochs pass without overall improvement.

```
net = torch.nn.Sequential(  
    torch.nn.Linear(32*32*3, 100),  
    torch.nn.ReLU(),  
    torch.nn.Linear(100, 100),  
    torch.nn.ReLU(),  
    torch.nn.Linear(100, 10)  
)
```

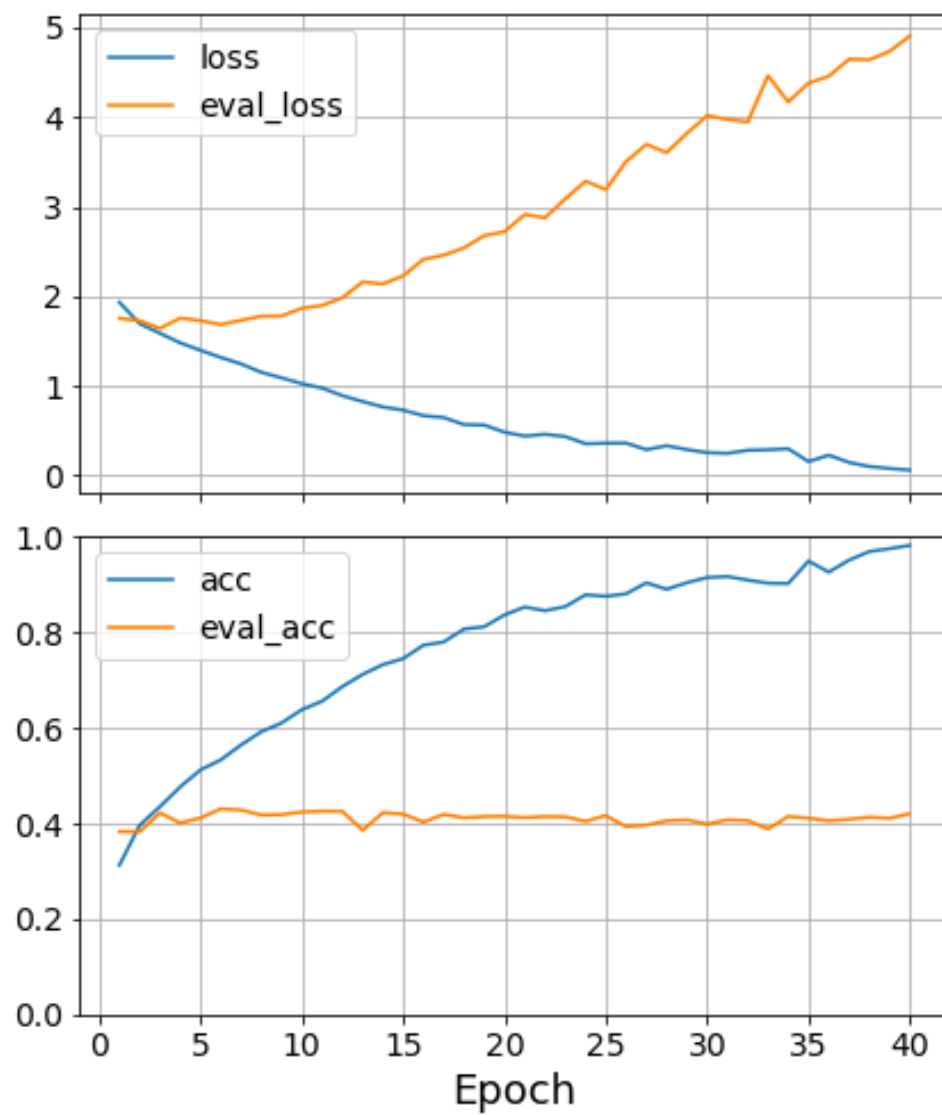
```
model = Model(net)
```

```
model.compile(optimizer = torch.optim.SGD(model.net.parameters(), lr=0.1),  
              loss = torch.nn.CrossEntropyLoss(),  
              metrics=[Accuracy()])
```

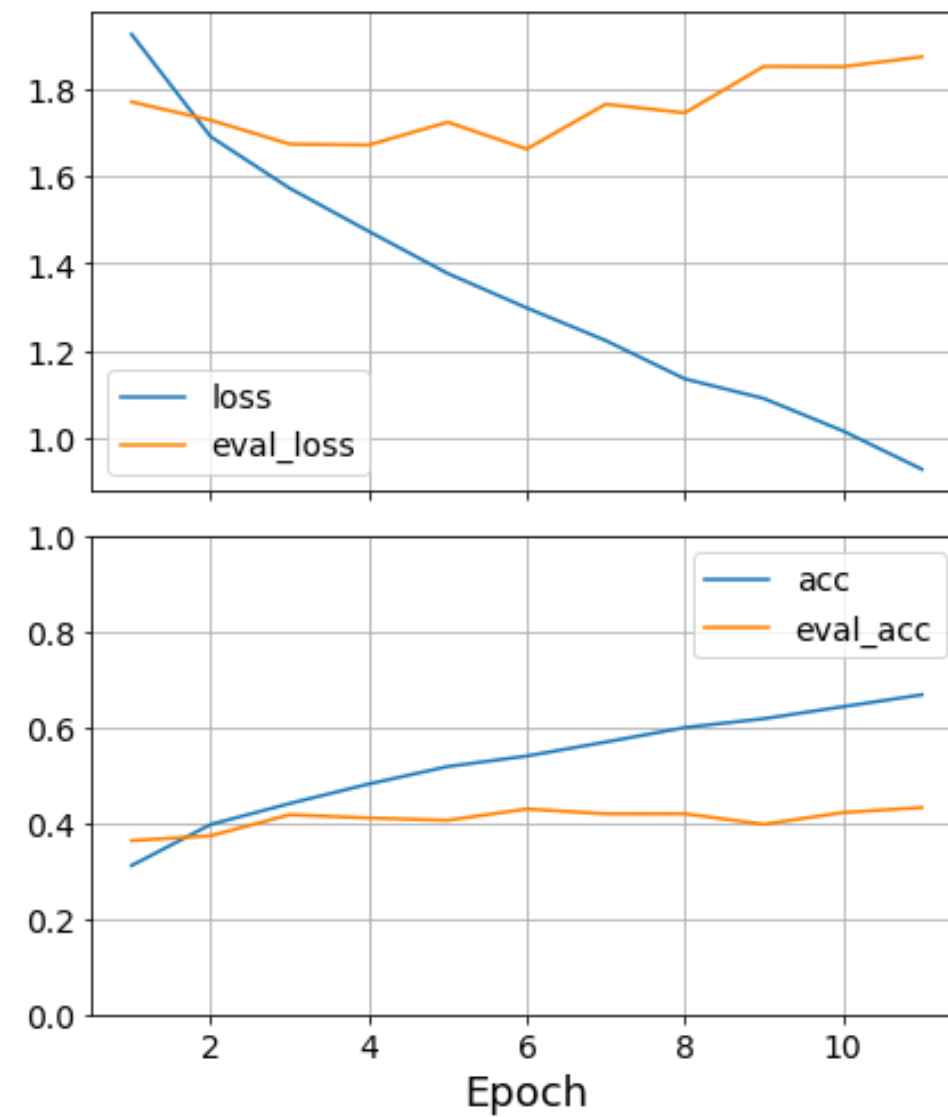
```
hist = model.fit(train_dataloader, eval_dataloader, epochs=40, early_stopping=5)
```

```
> Epoch 1/40 loss 1.92486 acc 0.3115 eval_loss 1.76986 eval_acc 0.3634  
> Epoch 2/40 loss 1.69015 acc 0.39725 eval_loss 1.72799 eval_acc 0.3737  
> Epoch 3/40 loss 1.57182 acc 0.44112 eval_loss 1.67308 eval_acc 0.4176  
> . . .  
> Epoch 10/40 loss 1.01528 acc 0.6435 eval_loss 1.85064 eval_acc 0.4222
```

No early stopping

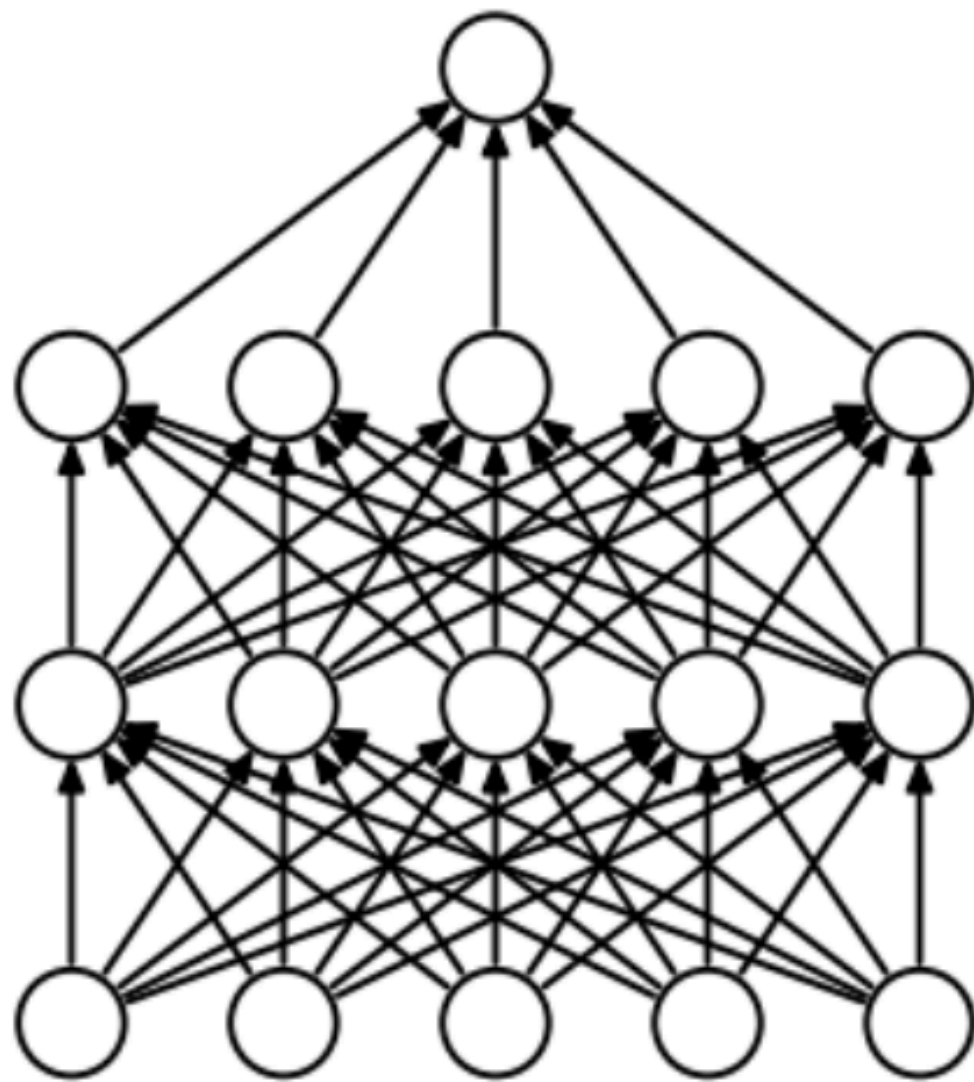


Early stopping

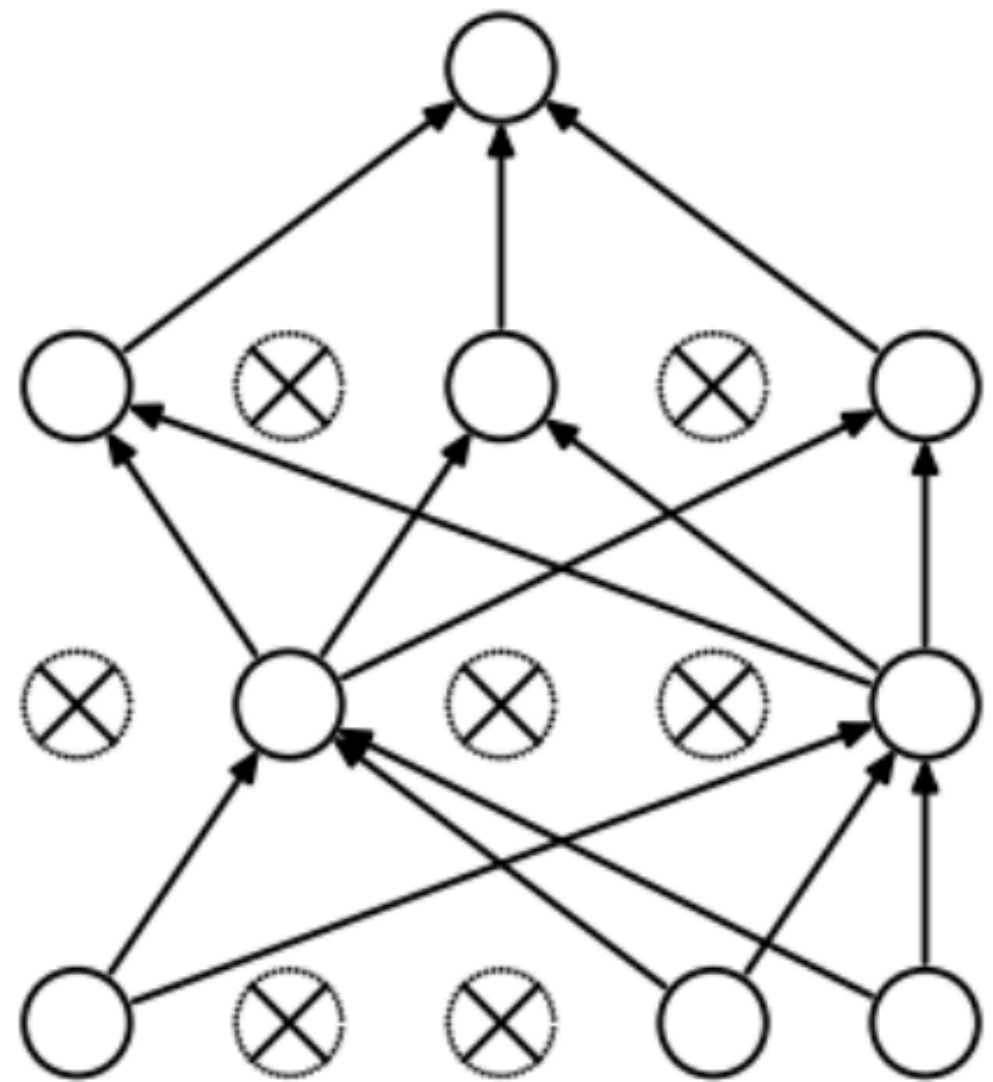


Dropout

- Popular technique to reduce overfitting. It consist on randomly “shut down” connexions during training with a given probability



(a) Standard Neural Net



(b) After applying dropout.

```
net = torch.nn.Sequential(
    torch.nn.Linear(32*32*3, 100),
    torch.nn.ReLU(),
    torch.nn.Dropout(p=0.5),
    torch.nn.Linear(100, 100),
    torch.nn.ReLU(),
    torch.nn.Dropout(p=0.3),
    torch.nn.Linear(100, 10)
)

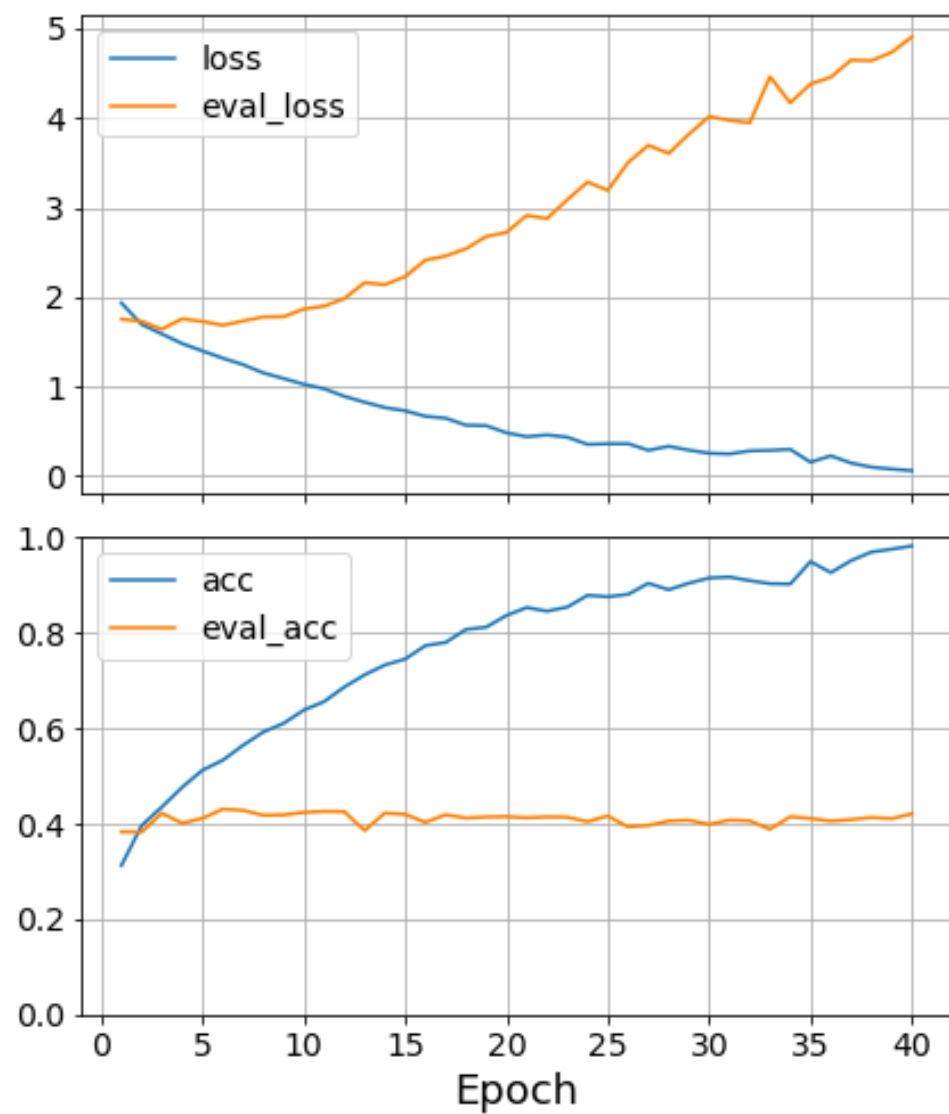
model = Model(net)

model.compile(optimizer = torch.optim.SGD(model.net.parameters(), lr=0.1),
              loss = torch.nn.CrossEntropyLoss(),
              metrics=[Accuracy()])

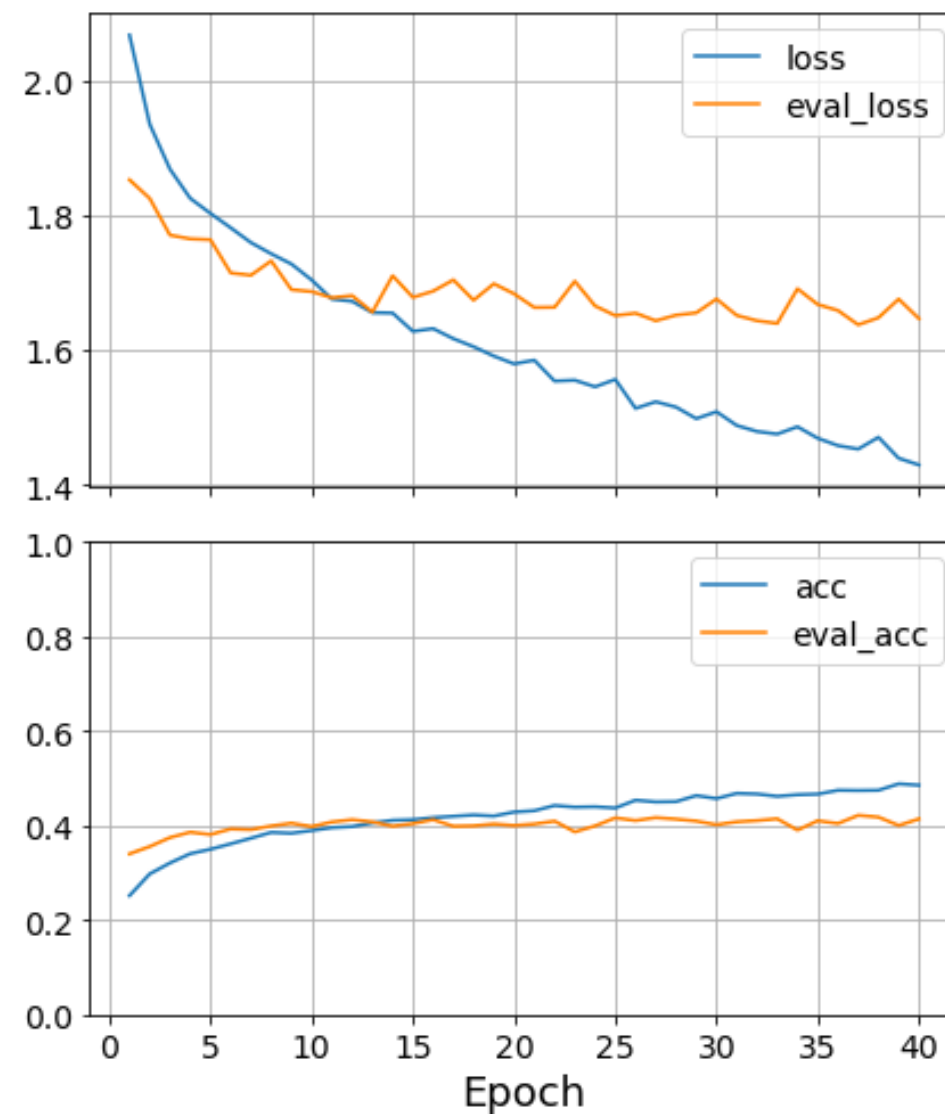
hist = model.fit(train_dataloader, eval_dataloader, epochs=40)

> Epoch 1/40 loss 2.06696 acc 0.25162 eval_loss 1.85249 eval_acc 0.34
> Epoch 2/40 loss 1.93478 acc 0.29762 eval_loss 1.82483 eval_acc 0.3555
> Epoch 3/40 loss 1.86819 acc 0.32075 eval_loss 1.77043 eval_acc 0.3745
> . . .
> Epoch 38/40 loss 1.47084 acc 0.4745 eval_loss 1.64803 eval_acc 0.4177
> Epoch 39/40 loss 1.43975 acc 0.48812 eval_loss 1.67565 eval_acc 0.4001
> Epoch 40/40 loss 1.42985 acc 0.48525 eval_loss 1.64681 eval_acc 0.4139
```

No Dropout



Dropout



Use more data

- The most effective way to reduce overfitting is using more data. Of course, this is not always possible.

```
train_dataset = Dataset(train_images, (mean, std), train_labels)
train_dataloader = DataLoader(train_dataset, shuffle=True, batch_size=batch_size)
```

```
len(train_dataset)
```

```
> 40000
```

```
net = torch.nn.Sequential(
    torch.nn.Linear(32*32*3, 100),
    torch.nn.ReLU(),
    torch.nn.Linear(100, 100),
    torch.nn.ReLU(),
    torch.nn.Linear(100, 10)
)
```

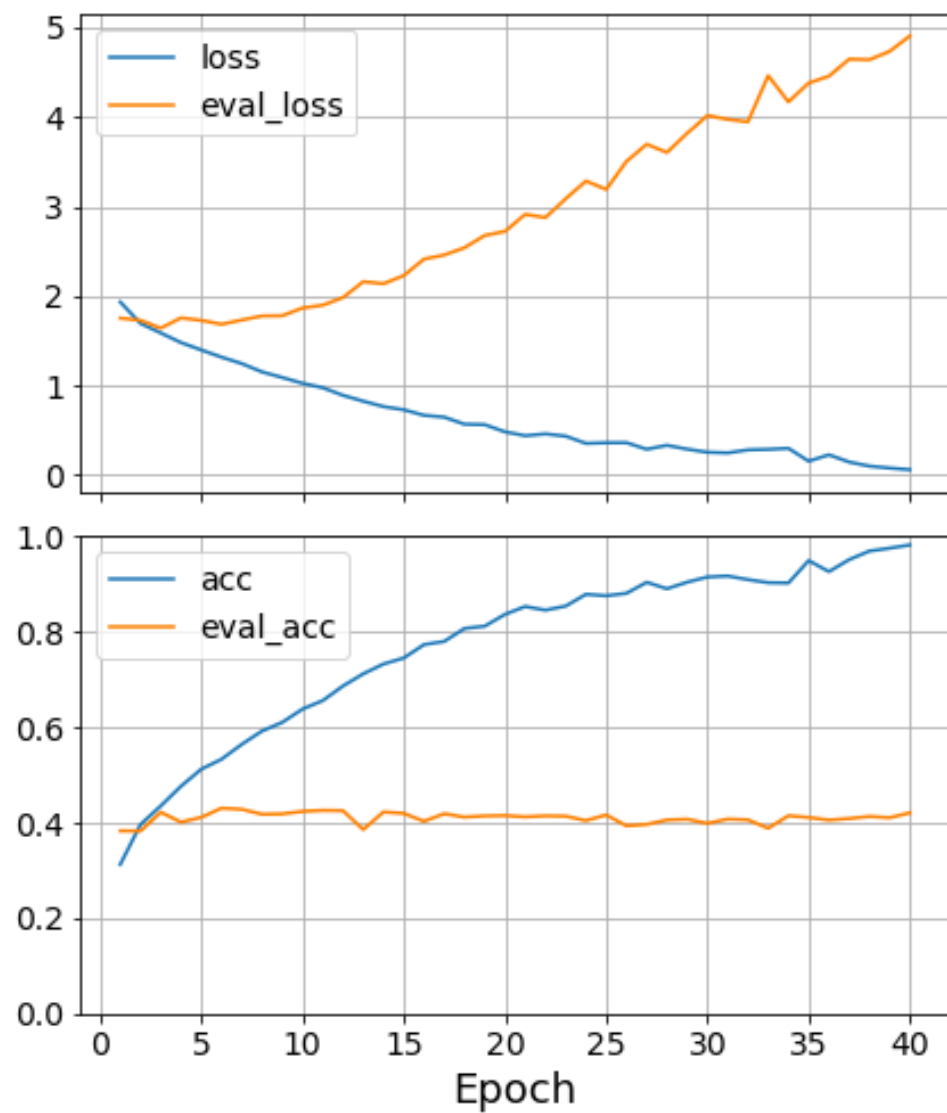
```
model = Model(net)
```

```
model.compile(optimizer = torch.optim.SGD(model.net.parameters(), lr=0.1),
              loss = torch.nn.CrossEntropyLoss(),
              metrics=[Accuracy()])
```

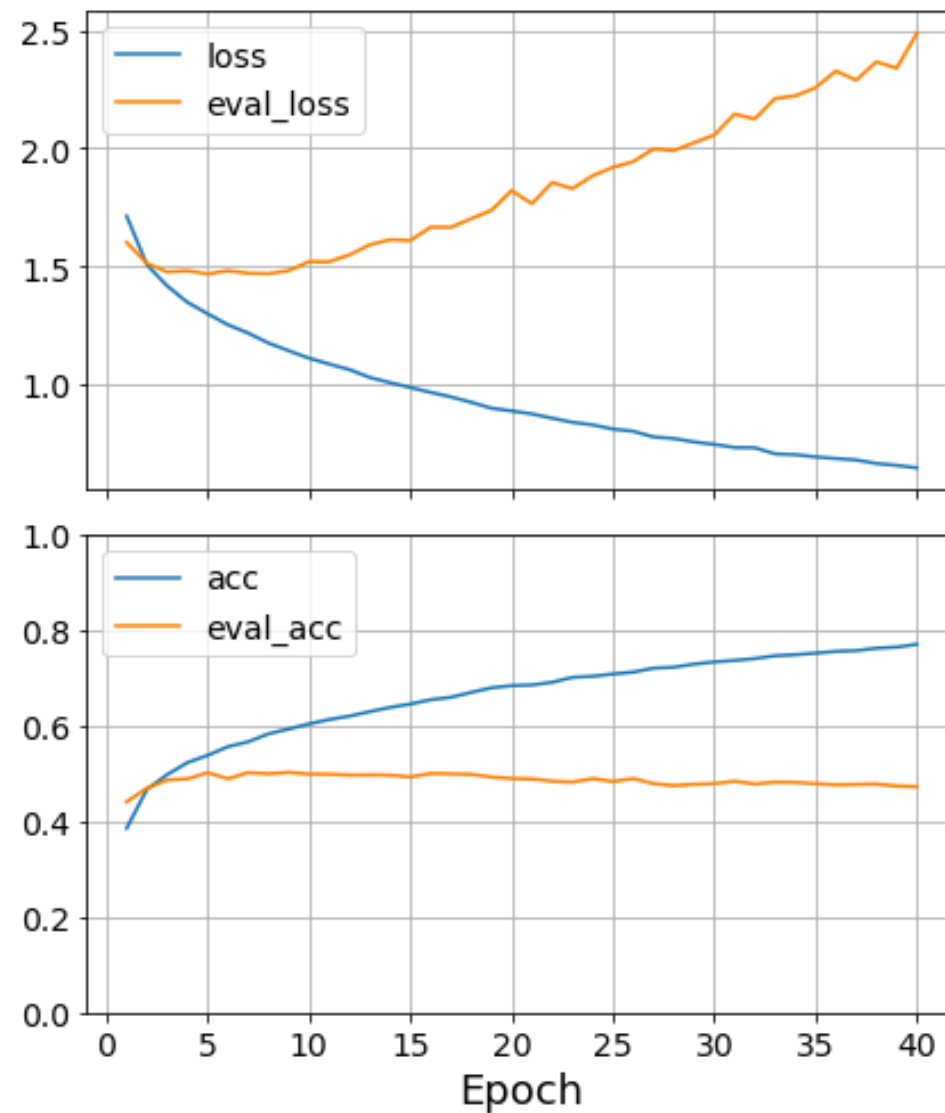
```
hist = model.fit(train_dataloader, eval_dataloader, epochs=40)
```

```
> Epoch 1/40 loss 1.71392 acc 0.38568 eval_loss 1.60215 eval_acc 0.4412
> Epoch 2/40 loss 1.50728 acc 0.46695 eval_loss 1.51316 eval_acc 0.4696
> Epoch 3/40 loss 1.41778 acc 0.498 eval_loss 1.4772 eval_acc 0.4868
> . . .
> Epoch 38/40 loss 0.66575 acc 0.76242 eval_loss 2.36665 eval_acc 0.4783
> Epoch 39/40 loss 0.65746 acc 0.76468 eval_loss 2.34025 eval_acc 0.4741
> Epoch 40/40 loss 0.64724 acc 0.7705 eval_loss 2.48766 eval_acc 0.4728
```


Data subset



All data

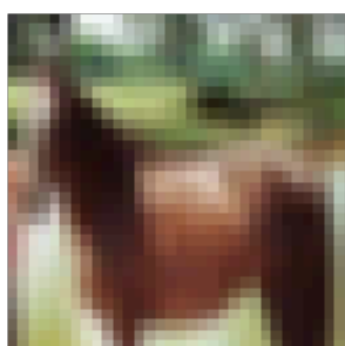
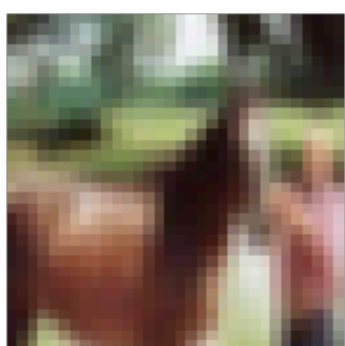
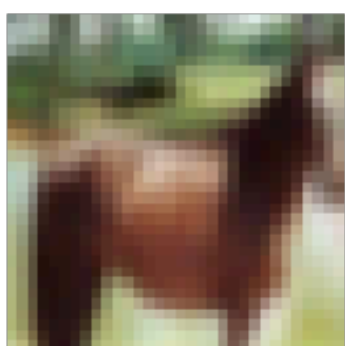
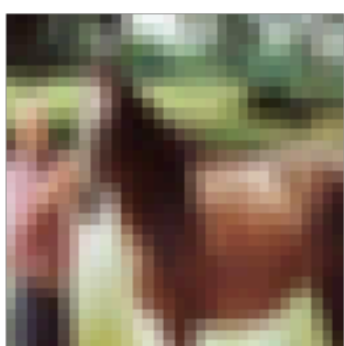
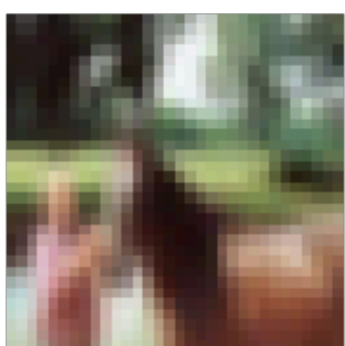
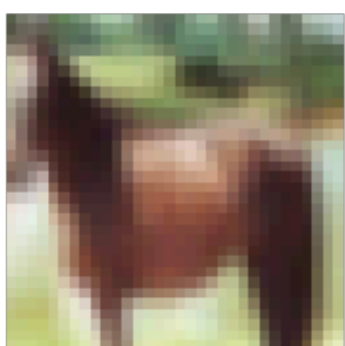
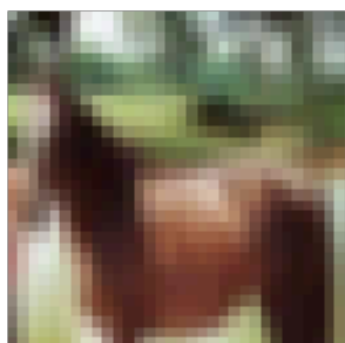
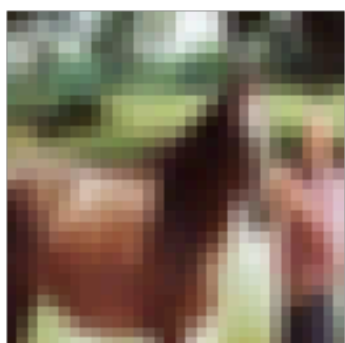
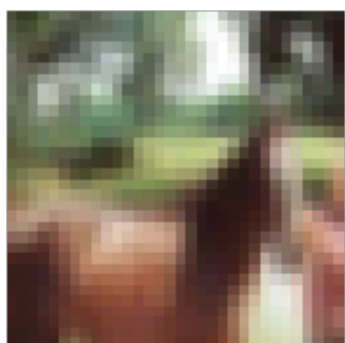
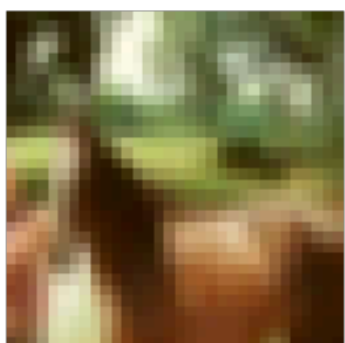
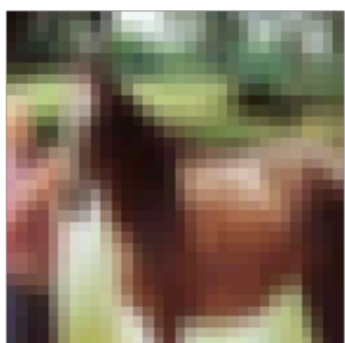
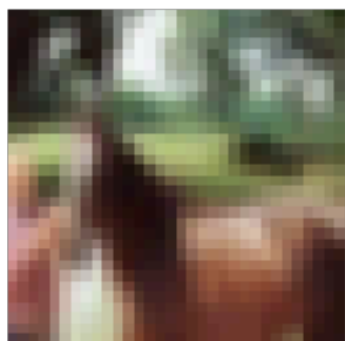
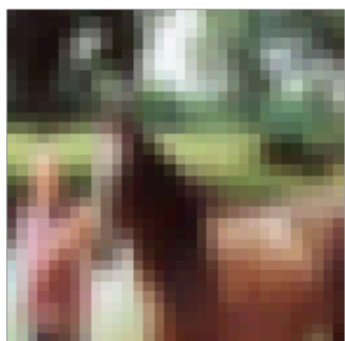
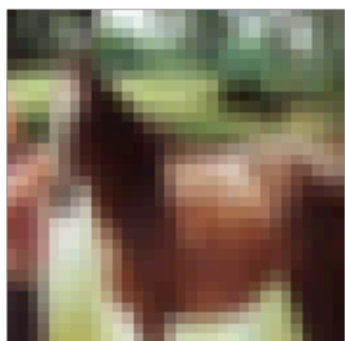
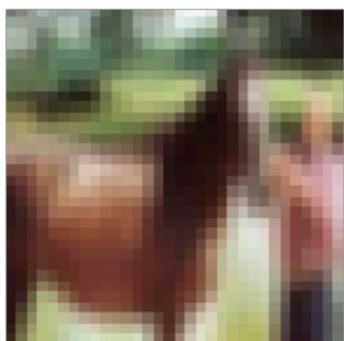
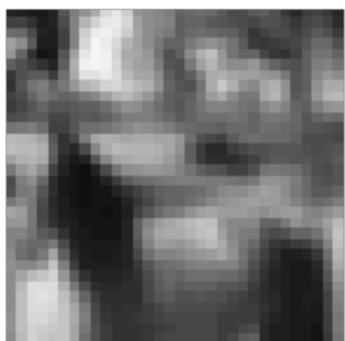
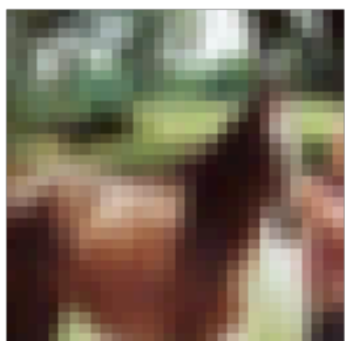


Data Augmentation

- Data augmentation is a technique for artificially increasing the size of our dataset by applying random transformations to our data before we give it to the network. We can virtually have an infinite dataset.

```
from augmentations import Compose, RandomCrop, Resize,
HorizontalFlip, ToGray, RGBShift, OneOf

trans = Compose([
    RandomCrop(24,24),
    Resize(32, 32),
    HorizontalFlip(),
    OneOf([
        ToGray(p=0.2),
        RGBShift(p=0.3)
    ])
])
```



```
class Dataset(torch.utils.data.Dataset):
    def __init__(self, X, stats, y=None, train=True, trans=None):
        self.X = X
        self.mean, self.std = stats
        self.y = y
        self.train = train
        self.trans = trans

    def __len__(self):
        return len(self.X)

    def __getitem__(self, ix):
        img = self.X[ix]
        if self.trans:
            img = self.trans(image=img) ["image"]
        img = torch.from_numpy(img / 255.)
        img = (img - self.mean) / self.std
        img = img.view(-1).float()
        if self.train:
            label = torch.tensor(self.y[ix]).long()
            return img, label
        return img
```

```
net = torch.nn.Sequential(
    torch.nn.Linear(32*32*3, 100),
    torch.nn.ReLU(),
    torch.nn.Linear(100, 100),
    torch.nn.ReLU(),
    torch.nn.Linear(100, 10)
)

model = Model(net)

model.compile(optimizer = torch.optim.SGD(model.net.parameters(), lr=0.1),
              loss = torch.nn.CrossEntropyLoss(),
              metrics=[Accuracy()])

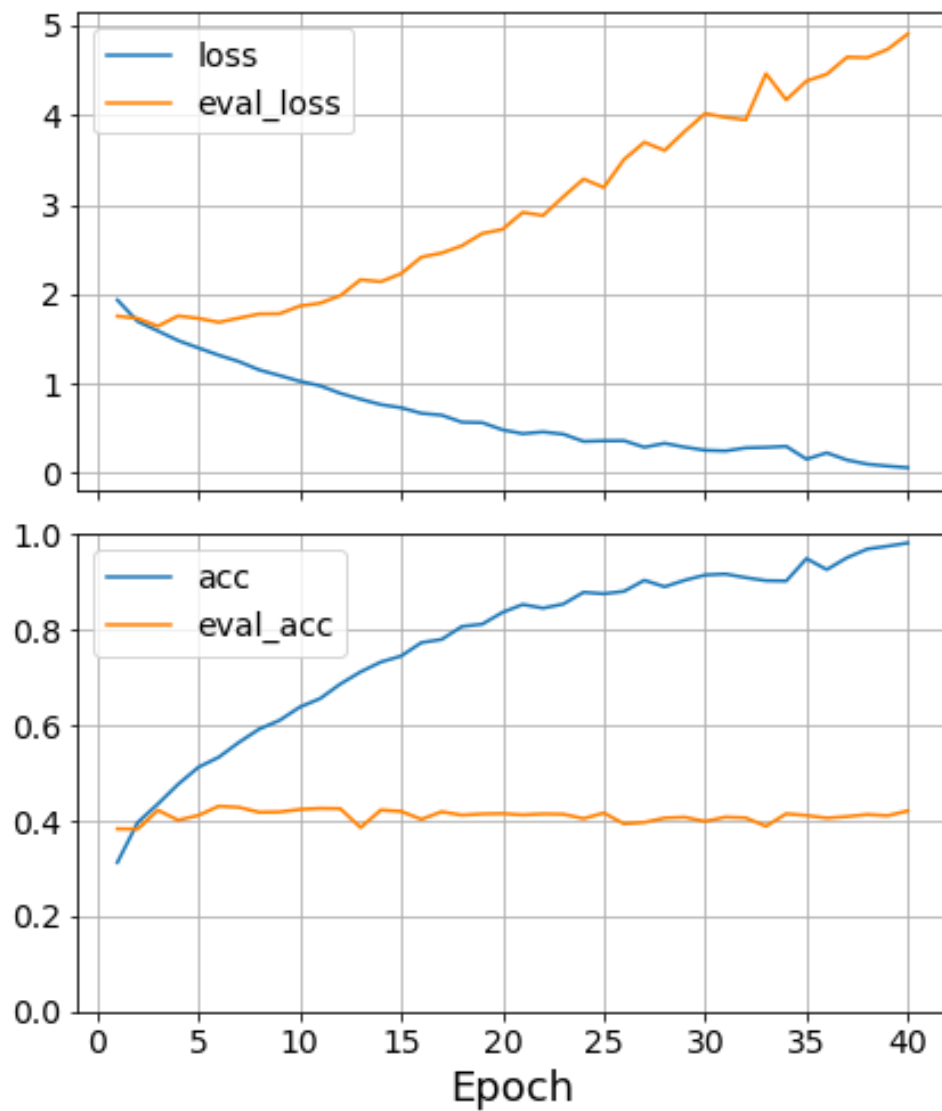
hist = model.fit(train_data_loader, eval_data_loader, epochs=40)

> Epoch 1/40 loss 2.07025 acc 0.239 eval_loss 1.90828 eval_acc 0.3165
> Epoch 2/40 loss 1.9492 acc 0.29488 eval_loss 1.84445 eval_acc 0.3387
> Epoch 3/40 loss 1.9078 acc 0.3045 eval_loss 1.8817 eval_acc 0.3407
> . . .
> Epoch 38/40 loss 1.59089 acc 0.4285 eval_loss 1.62051 eval_acc 0.4288
> Epoch 39/40 loss 1.60144 acc 0.421 eval_loss 1.66062 eval_acc 0.4226
> Epoch 40/40 loss 1.58869 acc 0.42738 eval_loss 1.65695 eval_acc 0.4283

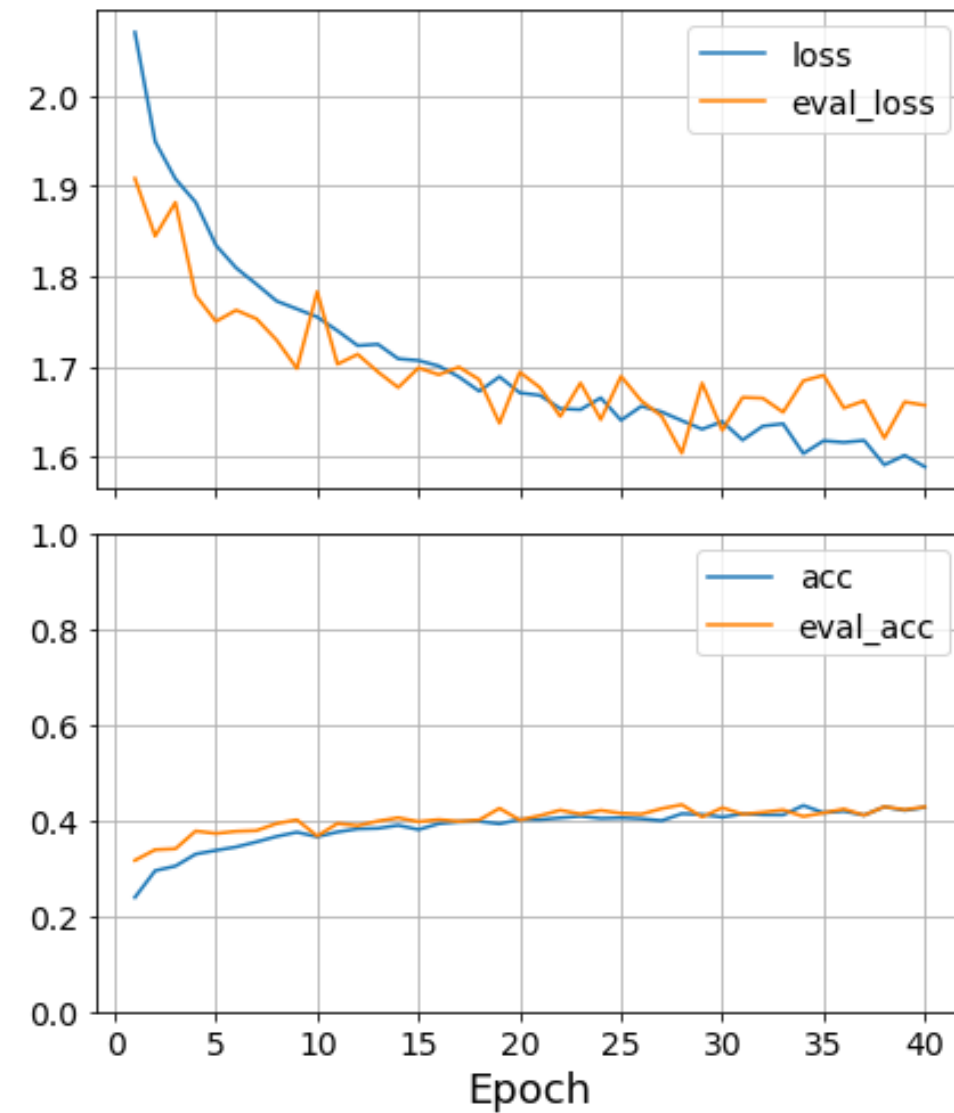
model.evaluate(test_data_loader)

> eval_loss 1.64193 eval_acc 0.42834
```

No data augmentation



Data augmentation

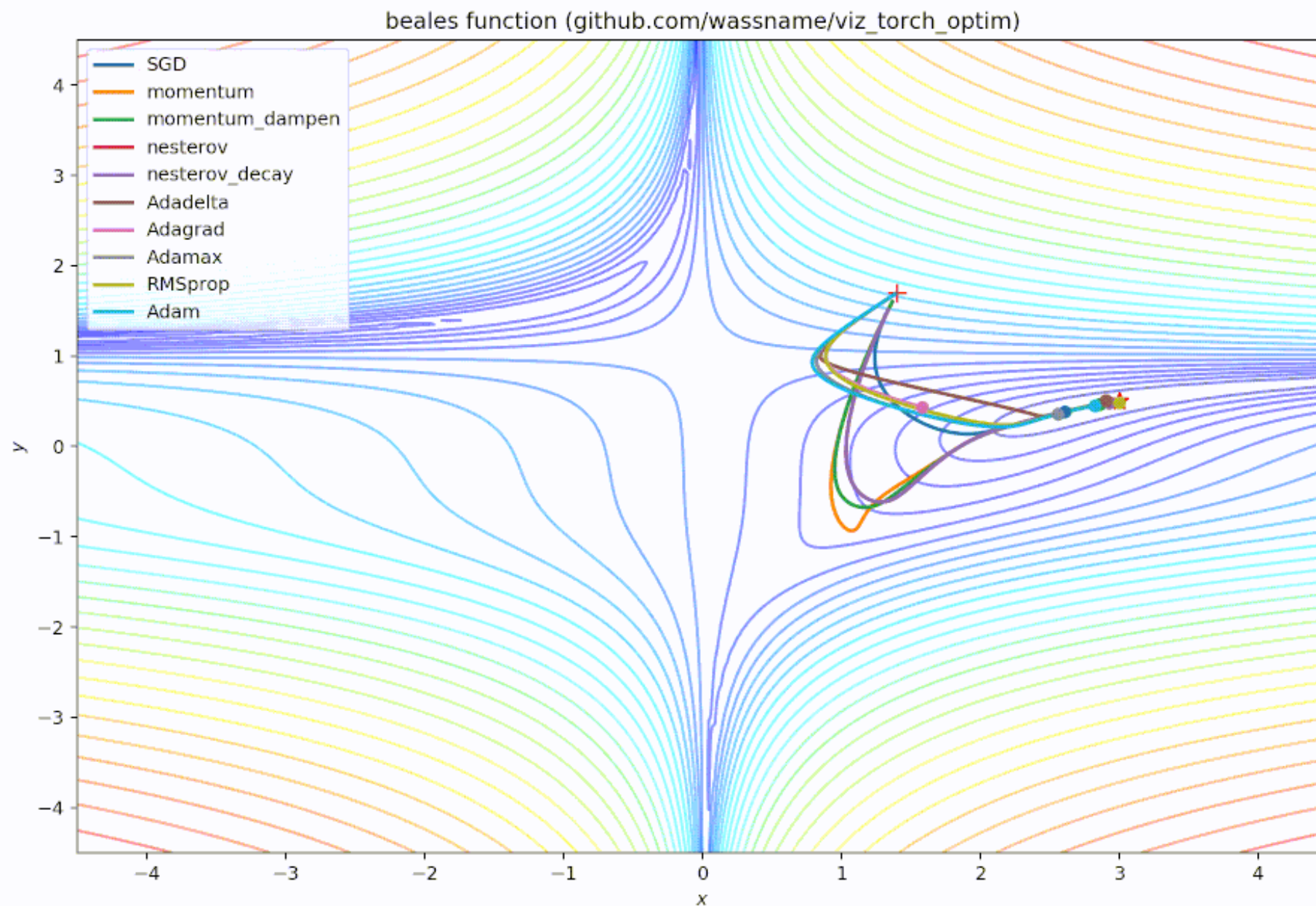


Faster Training

Faster training

- Faster optimizers
- Learning rate scheduling
- Batch Normalization
- Transfer learning

Faster optimizers



SGD with Momentum

- Remember the update rule of SGD: $w \leftarrow w - \eta \frac{\partial l}{\partial w}$
- We can accelerate training by adding a momentum term

$$m \leftarrow \beta m - \eta \frac{\partial l}{\partial w}$$

$$w \leftarrow w + m$$

Where β is a constant (0.9-0.99).

```
net = torch.nn.Sequential(
    torch.nn.Linear(32*32*3, 100),
    torch.nn.ReLU(),
    torch.nn.Linear(100, 100),
    torch.nn.ReLU(),
    torch.nn.Linear(100, 10)
)

model = Model(net)

model.compile(optimizer = torch.optim.SGD(model.net.parameters(), lr=0.01,
momentum=0.9),
              loss = torch.nn.CrossEntropyLoss(),
              metrics=[Accuracy()])

hist = model.fit(train_data_loader, eval_data_loader, epochs=40)

> Epoch 1/40 loss 2.09377 acc 0.23538 eval_loss 1.91709 eval_acc 0.3073
> Epoch 2/40 loss 1.95174 acc 0.28575 eval_loss 1.84105 eval_acc 0.3366
> Epoch 3/40 loss 1.90818 acc 0.30562 eval_loss 1.79173 eval_acc 0.3605
> . . .
> Epoch 38/40 loss 1.60942 acc 0.42162 eval_loss 1.63075 eval_acc 0.4276
> Epoch 39/40 loss 1.59842 acc 0.42425 eval_loss 1.60449 eval_acc 0.4255
> Epoch 40/40 loss 1.59501 acc 0.42812 eval_loss 1.63623 eval_acc 0.4274

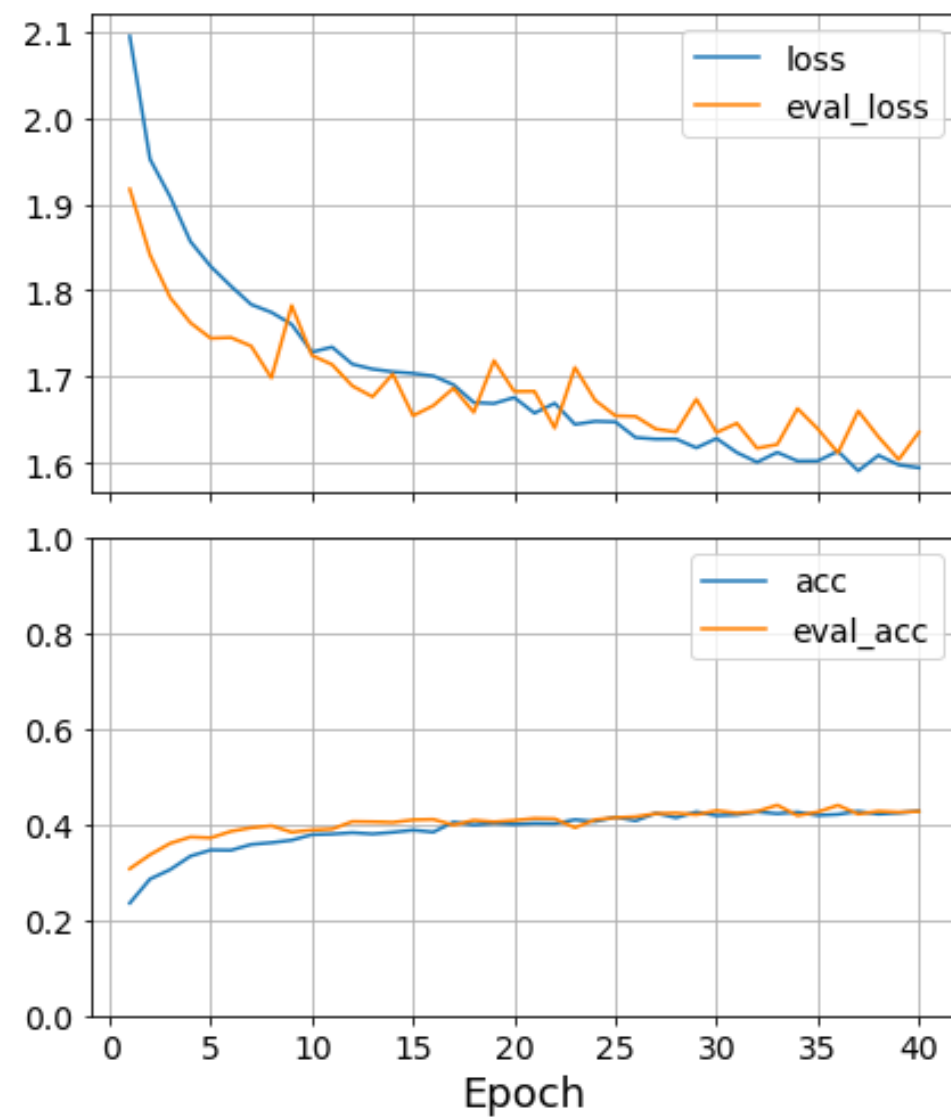
model.evaluate(test_data_loader)

> eval_loss 1.62791 eval_acc 0.42655
```

DA



DA + Momentum



RMSpop

Gradient Descent works by quickly going down the steepest slope, which may not point straight toward the global optimum. We can account for this by scaling down the gradient vector along the steepest dimensions.

$$s \leftarrow \beta s + (1 - \beta) \frac{\partial l}{\partial w} \otimes \frac{\partial l}{\partial w}$$
$$w \leftarrow w - \eta \frac{\partial l}{\partial w} \oslash \sqrt{s + \epsilon}$$

where β is the decay rate (normally 0.9) and ϵ is a smoothing term to avoid division by zero.

```
net = torch.nn.Sequential(
    torch.nn.Linear(32*32*3, 100),
    torch.nn.ReLU(),
    torch.nn.Linear(100, 100),
    torch.nn.ReLU(),
    torch.nn.Linear(100, 10)
)
```

```
model = Model(net)
```

```
model.compile(optimizer = torch.optim.RMSprop(model.net.parameters(), lr=0.001),
              loss = torch.nn.CrossEntropyLoss(),
              metrics=[Accuracy()])
```

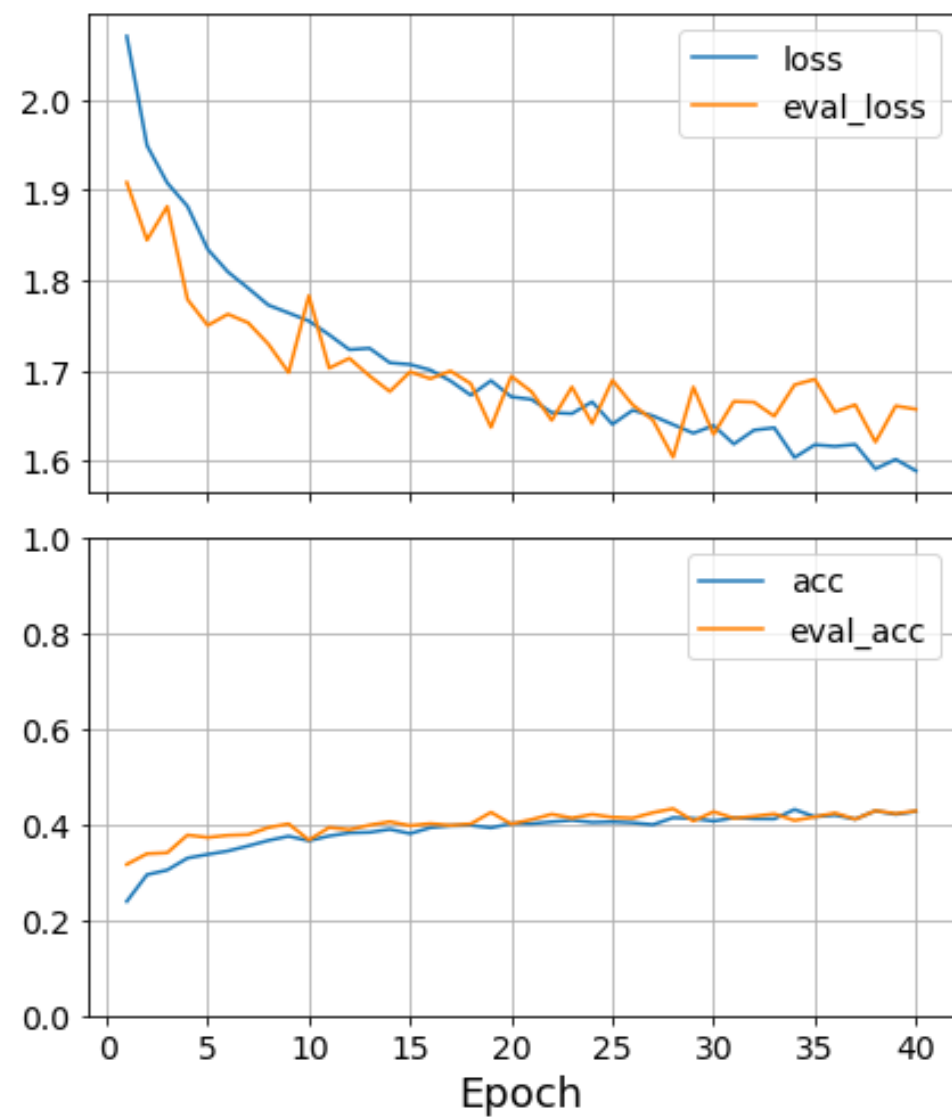
```
hist = model.fit(train_data_loader, eval_data_loader, epochs=40)
```

```
> Epoch 1/40 loss 2.12622 acc 0.249 eval_loss 1.92576 eval_acc 0.2958
> Epoch 2/40 loss 1.96298 acc 0.28625 eval_loss 1.89266 eval_acc 0.3287
> Epoch 3/40 loss 1.89255 acc 0.32325 eval_loss 1.79964 eval_acc 0.3628
> . . .
> Epoch 38/40 loss 1.59583 acc 0.435 eval_loss 1.68441 eval_acc 0.4155
> Epoch 39/40 loss 1.59173 acc 0.42975 eval_loss 1.63598 eval_acc 0.4289
> Epoch 40/40 loss 1.59811 acc 0.42912 eval_loss 1.64708 eval_acc 0.4165
```

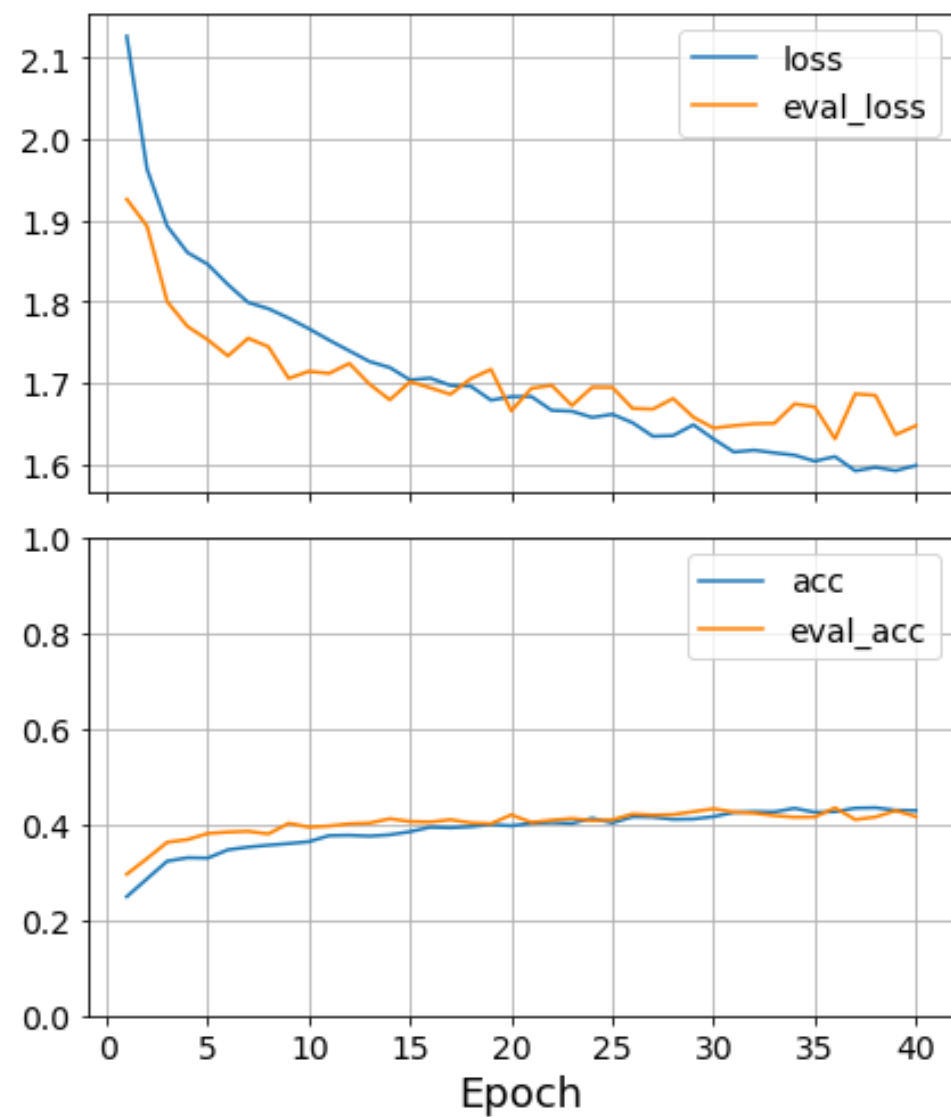
```
model.evaluate(test_data_loader)
```

```
> eval_loss 1.6422 eval_acc 0.42725
```

DA



DA + RMSprop



Adam

Adam combines ideas from momentum optimization and RMSprop, keeping track of an exponentially decaying average of past gradients and also an exponentially decaying average of past squared gradients.

$$m \leftarrow \beta_1 m + (1 - \beta_1) \frac{\partial l}{\partial w}$$

$$s \leftarrow \beta_2 s + (1 - \beta_2) \frac{\partial l}{\partial w} \otimes \frac{\partial l}{\partial w}$$

$$\hat{m} \leftarrow \frac{m}{1 - \beta_1^T}$$

$$\hat{s} \leftarrow \frac{s}{1 - \beta_2^T}$$

$$w \leftarrow w + \eta \hat{m} \oslash \sqrt{\hat{s} + \epsilon}$$

```
net = torch.nn.Sequential(
    torch.nn.Linear(32*32*3, 100),
    torch.nn.ReLU(),
    torch.nn.Linear(100, 100),
    torch.nn.ReLU(),
    torch.nn.Linear(100, 10)
)

model = Model(net)

model.compile(optimizer = torch.optim.Adam(model.net.parameters(), lr=0.001),
              loss = torch.nn.CrossEntropyLoss(),
              metrics=[Accuracy()])

hist = model.fit(train_data_loader, eval_data_loader, epochs=40)

> Epoch 1/40 loss 2.04972 acc 0.262 eval_loss 1.97075 eval_acc 0.3036
> Epoch 2/40 loss 1.95072 acc 0.29688 eval_loss 1.83995 eval_acc 0.3403
> Epoch 3/40 loss 1.89674 acc 0.32088 eval_loss 1.80137 eval_acc 0.3679
> . . .
> Epoch 38/40 loss 1.58355 acc 0.438 eval_loss 1.63605 eval_acc 0.427
> Epoch 39/40 loss 1.59618 acc 0.42712 eval_loss 1.60884 eval_acc 0.4369
> Epoch 40/40 loss 1.58444 acc 0.4325 eval_loss 1.63319 eval_acc 0.4271

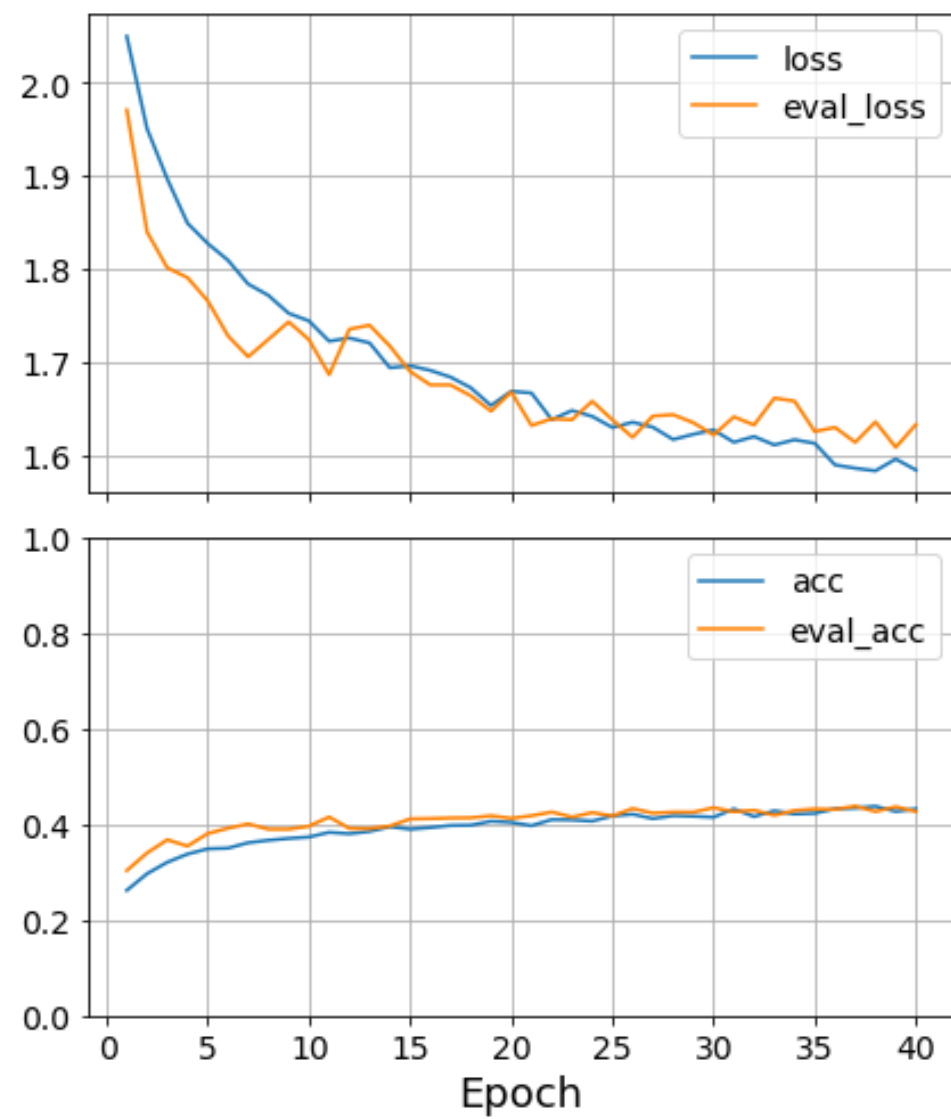
model.evaluate(test_data_loader)

eval_loss 1.61263 eval_acc 0.42874
```

DA

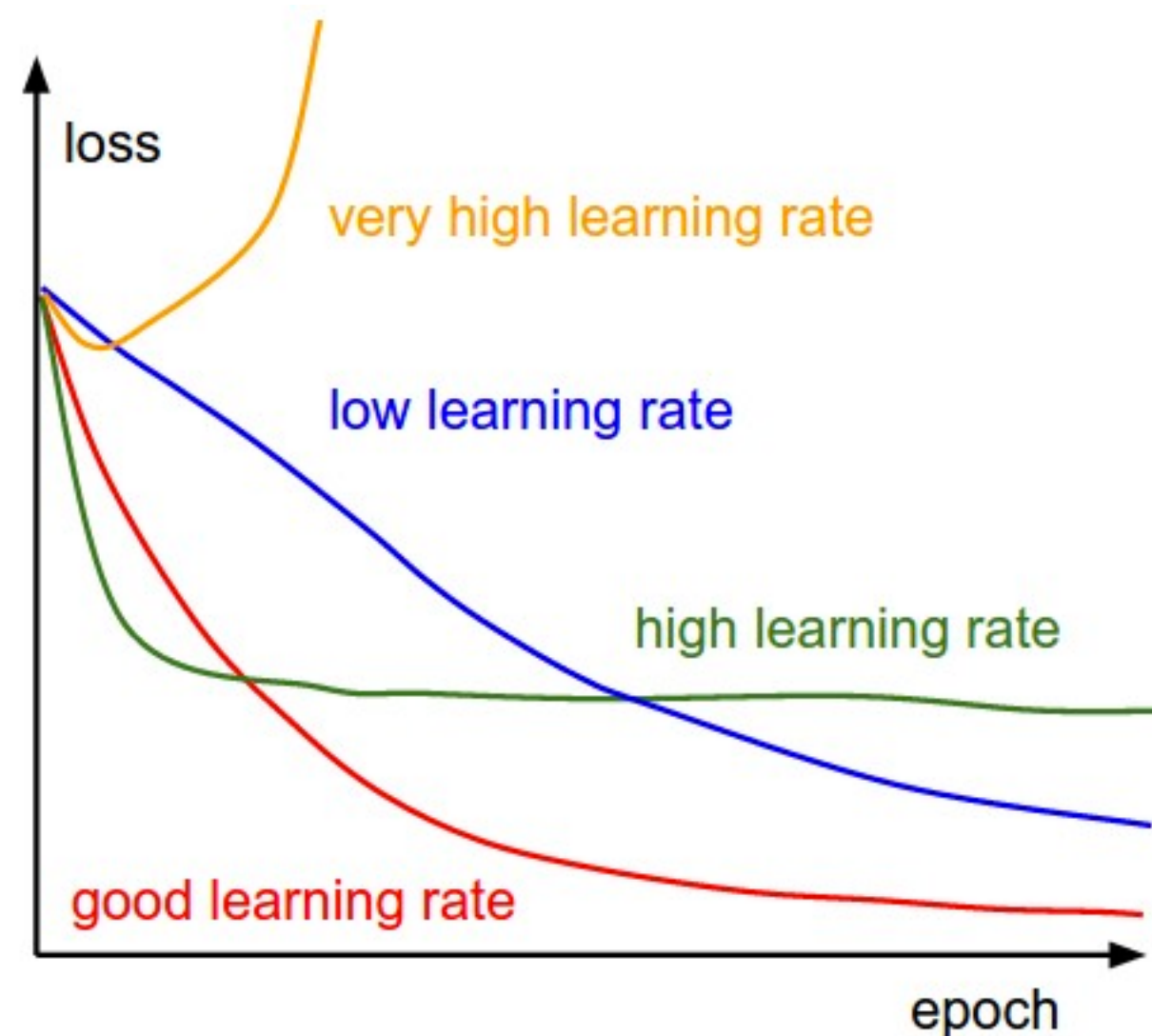


DA + Adam



Learning Rate Scheduling

We can also accelerate learning by changing the learning rate during training.




```

net = torch.nn.Sequential(
    torch.nn.Linear(32*32*3, 100),
    torch.nn.ReLU(),
    torch.nn.Linear(100, 100),
    torch.nn.ReLU(),
    torch.nn.Linear(100, 10)
)

model = Model(net)

optimizer = torch.optim.SGD(model.net.parameters(), lr=0.1)

# divide learning rate by 2 every 10 epochs
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, 10, 0.5)

model.compile(optimizer = optimizer,
              loss = torch.nn.CrossEntropyLoss(),
              scheduler = scheduler,
              metrics=[Accuracy()])

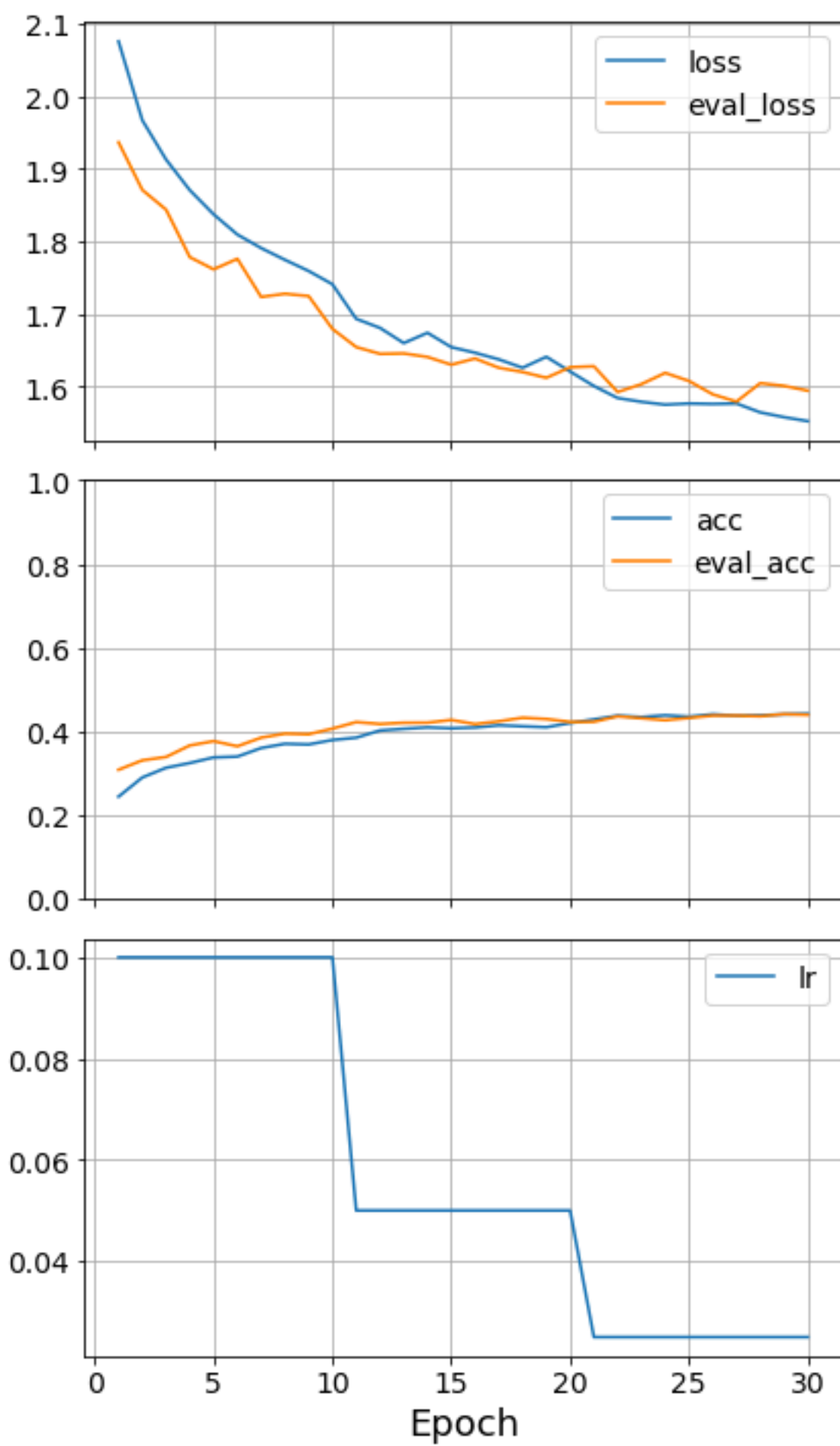
hist = model.fit(train_data_loader, eval_data_loader, epochs=30)

> Epoch 1/30 loss 2.07576 acc 0.24288 eval_loss 1.93678 eval_acc 0.3073
> Epoch 2/30 loss 1.96714 acc 0.28875 eval_loss 1.87116 eval_acc 0.3298
> Epoch 3/30 loss 1.91301 acc 0.312 eval_loss 1.84404 eval_acc 0.3378
> . . .
> Epoch 28/30 loss 1.56475 acc 0.4375 eval_loss 1.60497 eval_acc 0.4354
> Epoch 29/30 loss 1.55814 acc 0.44 eval_loss 1.60126 eval_acc 0.4405
> Epoch 30/30 loss 1.55271 acc 0.44125 eval_loss 1.59467 eval_acc 0.4388

model.evaluate(test_data_loader)

> eval_loss 1.59014 eval_acc 0.44397

```



Batch Normalization

This is a normalization technique that can accelerate training and also acts as a regularization method, reducing overfitting. It consists on a layer that zero-centers and normalizes each input and then scales and shifts the result using two parameters. Since these parameters are learnable, the network can decide the best way to scale and shift the layer's inputs to achieve its task.

$$\mu_B = \frac{1}{m_B} \sum_{i=1}^{m_B} x^{(i)}$$

$$\sigma_B^2 = \frac{1}{m_B} \sum_{i=1}^{m_B} (x^{(i)} - \mu_B)^2$$

$$\hat{x}^{(i)} = \frac{x^{(i)} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$z^{(i)} = \gamma \otimes \hat{x}^{(i)} + \beta$$

- μ_B : input means over mini-batch B .
- σ_B : input standard deviations over mini-batch.
- m_B : number of instances in the mini-batch
- $\hat{x}^{(i)}$: zero-centered and normalized inputs
- γ : scale parameter
- β : shift parameter
- $z^{(i)}$: output of the BN layer

```
net = torch.nn.Sequential(
    torch.nn.Linear(32*32*3, 100),
    torch.nn.ReLU(),
    torch.nn.BatchNorm1d(100),
    torch.nn.Linear(100, 100),
    torch.nn.ReLU(),
    torch.nn.BatchNorm1d(100),
    torch.nn.Linear(100, 10)
)
```

```
model = Model(net)
```

```
optimizer = torch.optim.SGD(model.net.parameters(), lr=0.1)
```

```
model.compile(optimizer = optimizer,
              loss = torch.nn.CrossEntropyLoss(),
              metrics=[Accuracy()])
```

```
hist = model.fit(train_data_loader, eval_data_loader, epochs=40)
```

```
> Epoch 1/40 loss 2.04087 acc 0.249 eval_loss 1.92968 eval_acc 0.2895
> Epoch 2/40 loss 1.91333 acc 0.30638 eval_loss 1.81082 eval_acc 0.3427
> Epoch 3/40 loss 1.86876 acc 0.31862 eval_loss 1.79373 eval_acc 0.346
> . . .
> Epoch 38/40 loss 1.55353 acc 0.443 eval_loss 1.56177 eval_acc 0.445
> Epoch 39/40 loss 1.56178 acc 0.43738 eval_loss 1.58352 eval_acc 0.4375
> Epoch 40/40 loss 1.56455 acc 0.44188 eval_loss 1.5498 eval_acc 0.4503
```

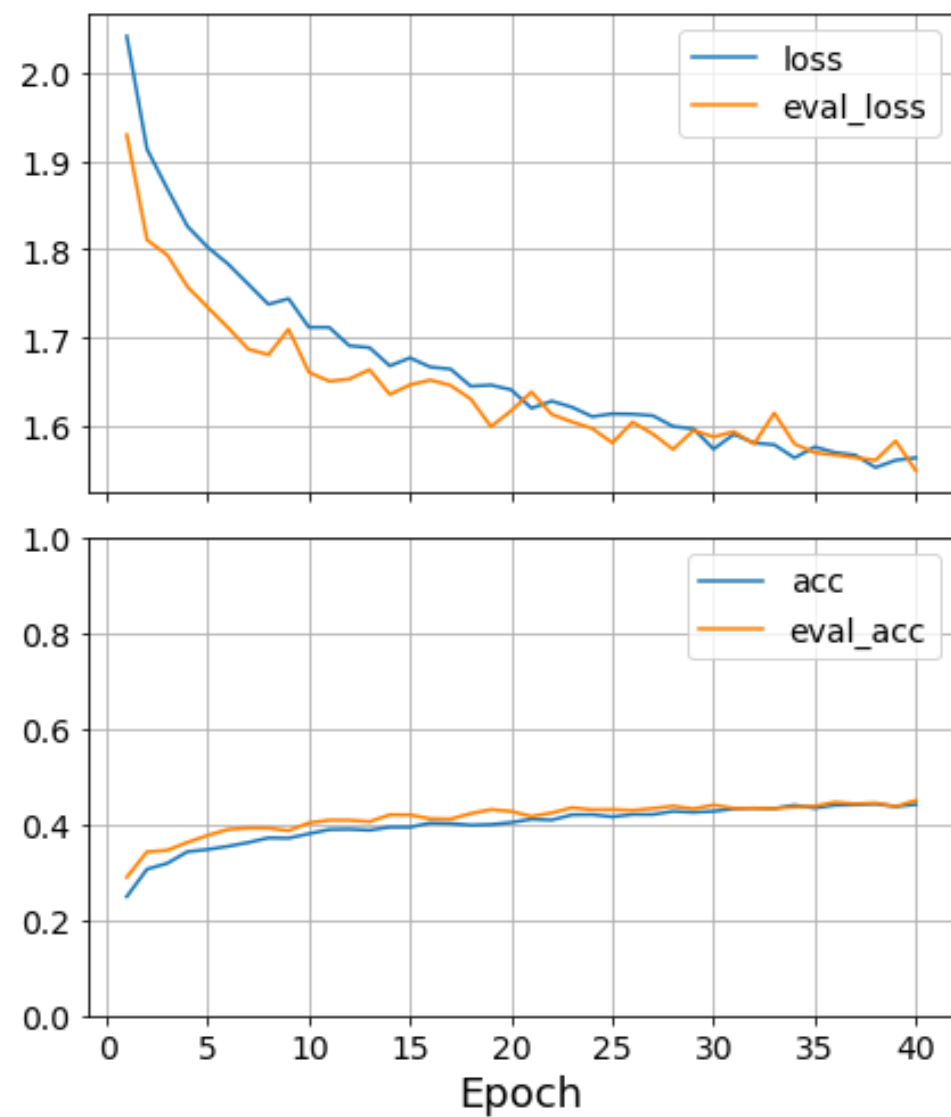
```
model.evaluate(test_data_loader)
```

```
> eval_loss 1.54788 eval_acc 0.44895
```

DA



DA + BN



Transfer learning

- Idea: Train a NN with one dataset and then use it for a different one.
- If the features learnt with the first dataset are “transferable” to the second task, we can avoid re-training from scratch,
- In Computer Vision, in practice we use networks pre-trained on large datasets (like Imagenet) and fine-tune them for our particular task.
- We will use this extensively from now on.

Putting everything together

- Weight decay (with SGD)
- Batch Norm / Dropout
- Data augmentation
- Learning Rate Scheduling / Early Stopping
- SGD with momentum / Adam / RMSprop
- Explore different combinations in subsampled dataset (plus batch size, learning rate, different networks, etc).
- Train final model with the best hyper parameter combination on all data

```

net = torch.nn.Sequential(
    torch.nn.Linear(32*32*3, 100),
    torch.nn.ReLU(),
    torch.nn.BatchNorm1d(100),
    torch.nn.Linear(100, 100),
    torch.nn.ReLU(),
    torch.nn.BatchNorm1d(100),
    torch.nn.Linear(100, 10)
)

optimizer = torch.optim.Adam(model.net.parameters())
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, 10, 0.5)

model.compile(optimizer = optimizer,
              loss = torch.nn.CrossEntropyLoss(),
              scheduler = scheduler,
              metrics=[Accuracy()])

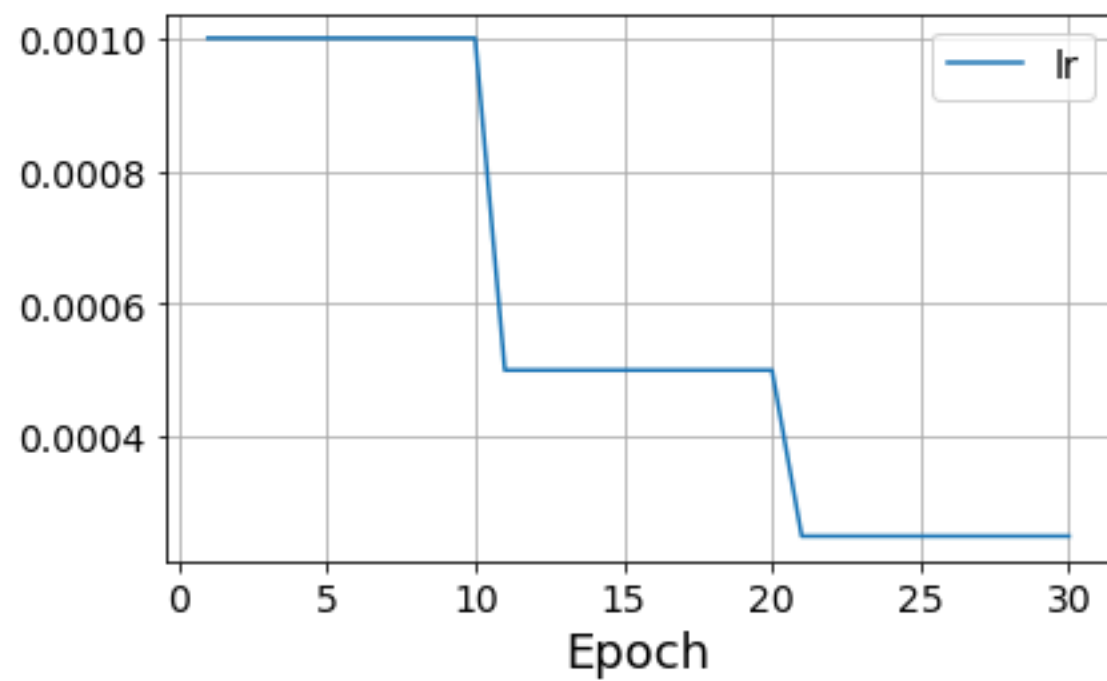
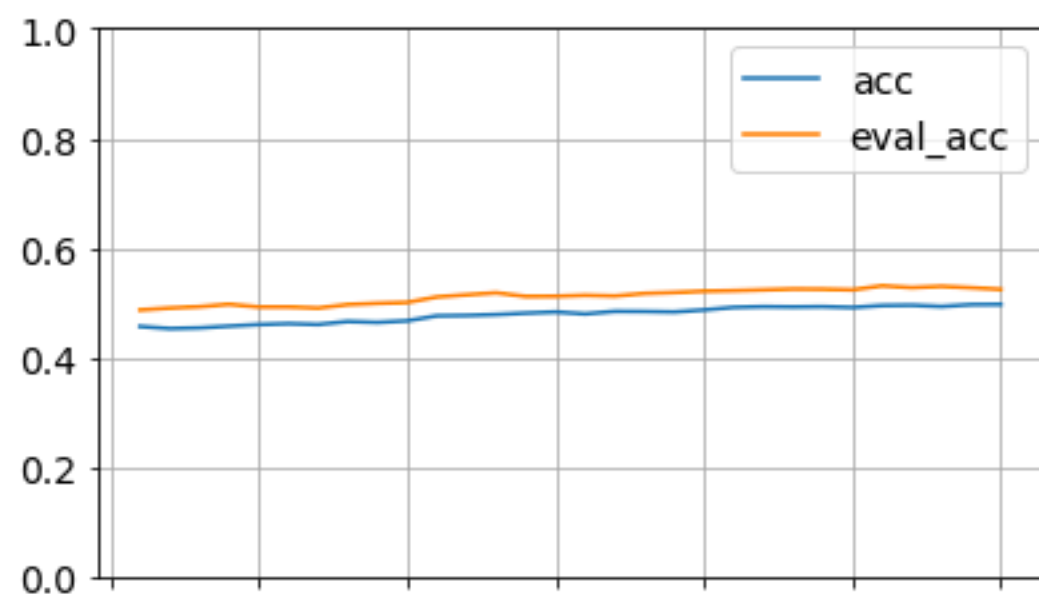
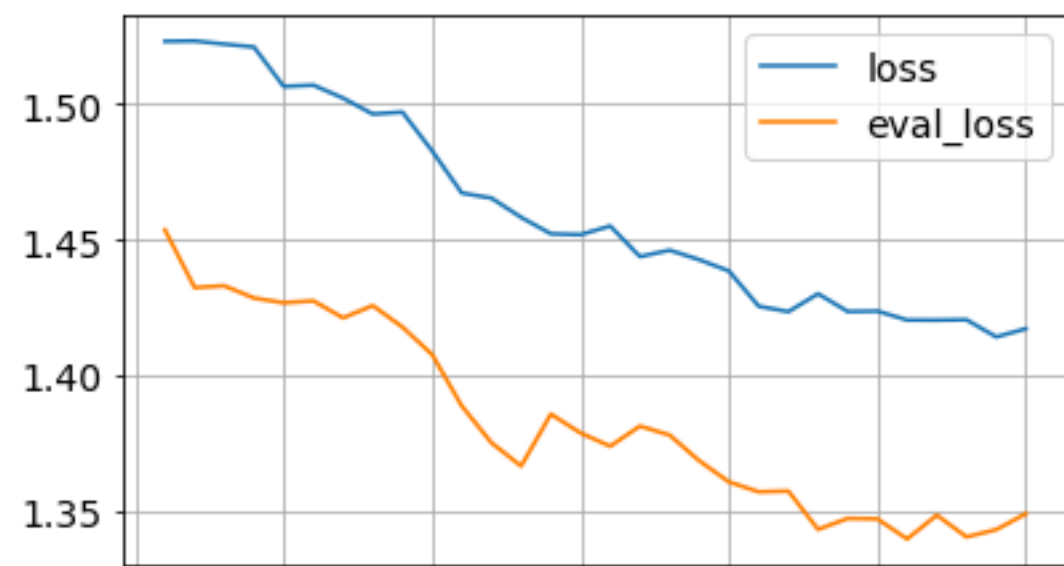
hist = model.fit(train_data_loader, eval_data_loader, epochs=30)

> Epoch 1/30 loss 1.52311 acc 0.45632 eval_loss 1.4537 eval_acc 0.4864
> Epoch 2/30 loss 1.52324 acc 0.45248 eval_loss 1.43243 eval_acc 0.49
> Epoch 3/30 loss 1.52208 acc 0.4537 eval_loss 1.43309 eval_acc 0.4924
> . . .
> Epoch 28/30 loss 1.4206 acc 0.49282 eval_loss 1.34063 eval_acc 0.5296
> Epoch 29/30 loss 1.41427 acc 0.49622 eval_loss 1.34331 eval_acc 0.527
> Epoch 30/30 loss 1.41728 acc 0.49658 eval_loss 1.34915 eval_acc 0.5241

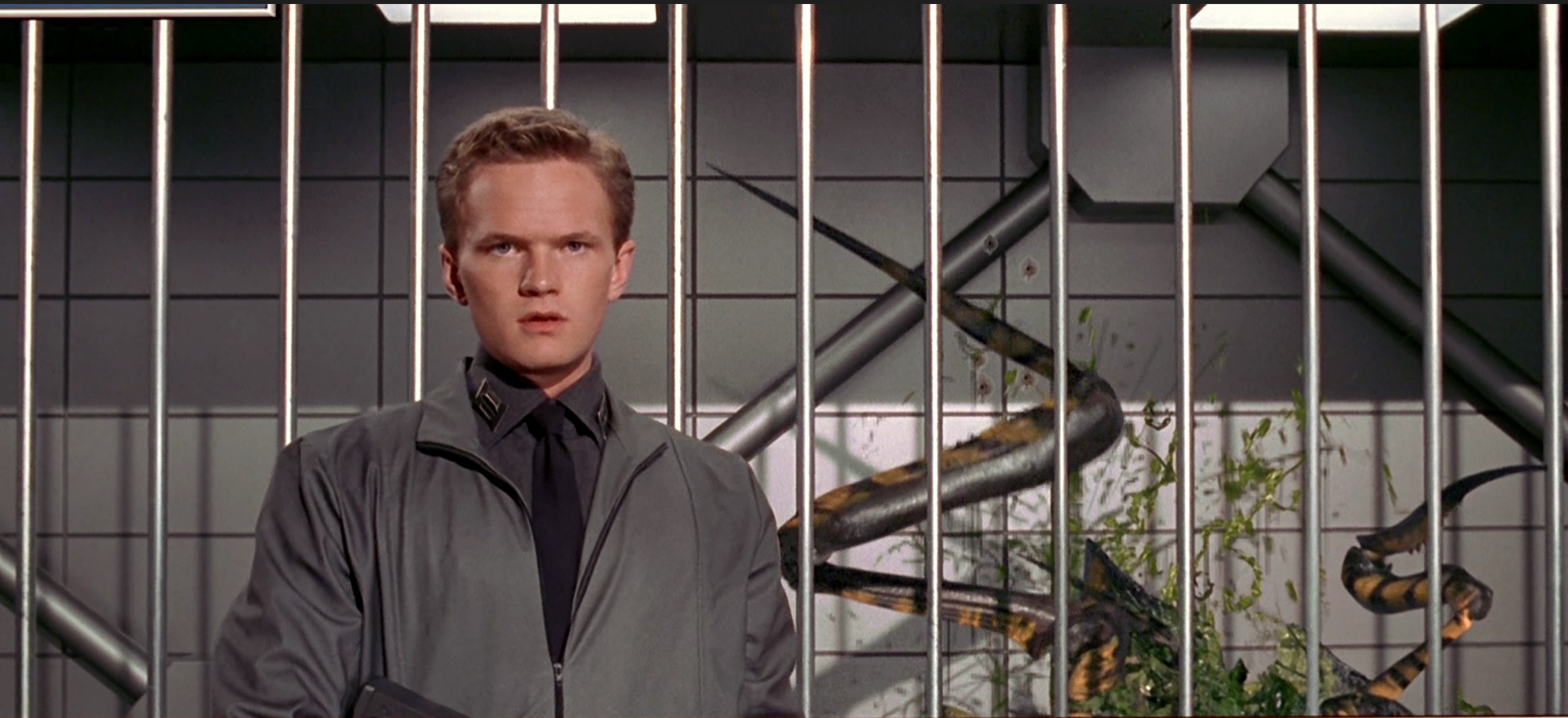
model.evaluate(test_data_loader)

> eval_loss 1.35423 eval_acc 0.5201

```

<https://github.com/sensioai/dl/blob/master/training/training.ipynb>



✚ WOULD YOU LIKE TO KNOW MORE?

Let's do a challenge !



<https://www.kaggle.com/t/5a581ca0d6a341f29e6afc024f0731e1>

The background of the slide features a bokeh effect with numerous out-of-focus circles in shades of yellow, orange, and blue against a dark navy blue background. The circles vary in size and opacity, creating a soft, glowing effect.

Training Neural Networks