# Introduction to Deep Learning (I2DL)

## Solution to exercise 3 and exam tips

# Today

- Solution of Exercise 3
  - Facial Keypoints
  - Recurrent Neural Networks
  - Segmentation
- Practical Pytorch Stuff
- Exam Topics and Mock Exam
- Organizationional Stuff

# Solutions to EX3

# Spatial Batchnormalization

- Forward

- Backward

```
424     ##############################################################################
425
426     # Computation in one sweep by rearranging the dims to fit into
427     # the batchnorm_forward framework
428     x_swapped = np.transpose(x, (0, 2, 3, 1))
429     x_swapped_reshaped = np.reshape(x_swapped, (-1, x_swapped.shape[-1]))
430
431     out_temp, cache = batchnorm_forward(
432         x_swapped_reshaped, gamma, beta, bn_param)
433     out = np.transpose(np.reshape(out_temp, x_swapped.shape), (0, 3, 1, 2))
434
435     ##############################################################################
```

```
463     ##############################################################################
464
465     dout_swapped = np.transpose(dout, (0, 2, 3, 1))
466     dout_swapped_reshaped = np.reshape(
467         dout_swapped, (-1, dout_swapped.shape[-1]))
468
469     dx_sr, dgamma, dbeta = batchnorm_backward(dout_swapped_reshaped, cache)
470
471     dx = np.transpose(np.reshape(dx_sr, dout_swapped.shape), (0, 3, 1, 2))
472
473     ##############################################################################
```

# A word of warning

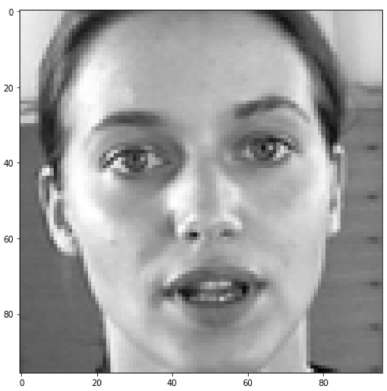
- Be aware of the differences of regular and spatial Batchnormalization and Dropout
- Pytorch will usually not throw an error as it is a valid possibility

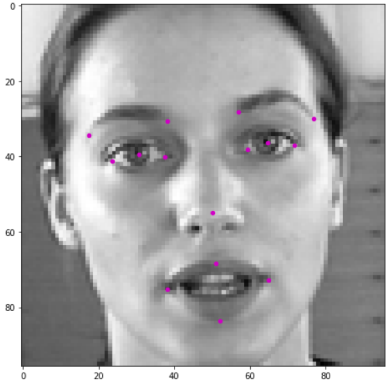
E.g., for 2d tasks use:

- Dropout2d
- BatchNorm2d

# Facial Keypoints

- Facial Keypoint extraction is a Regression task



Input Image

Coordinates Facial Keypoints

# Facial Keypoints Dataloader

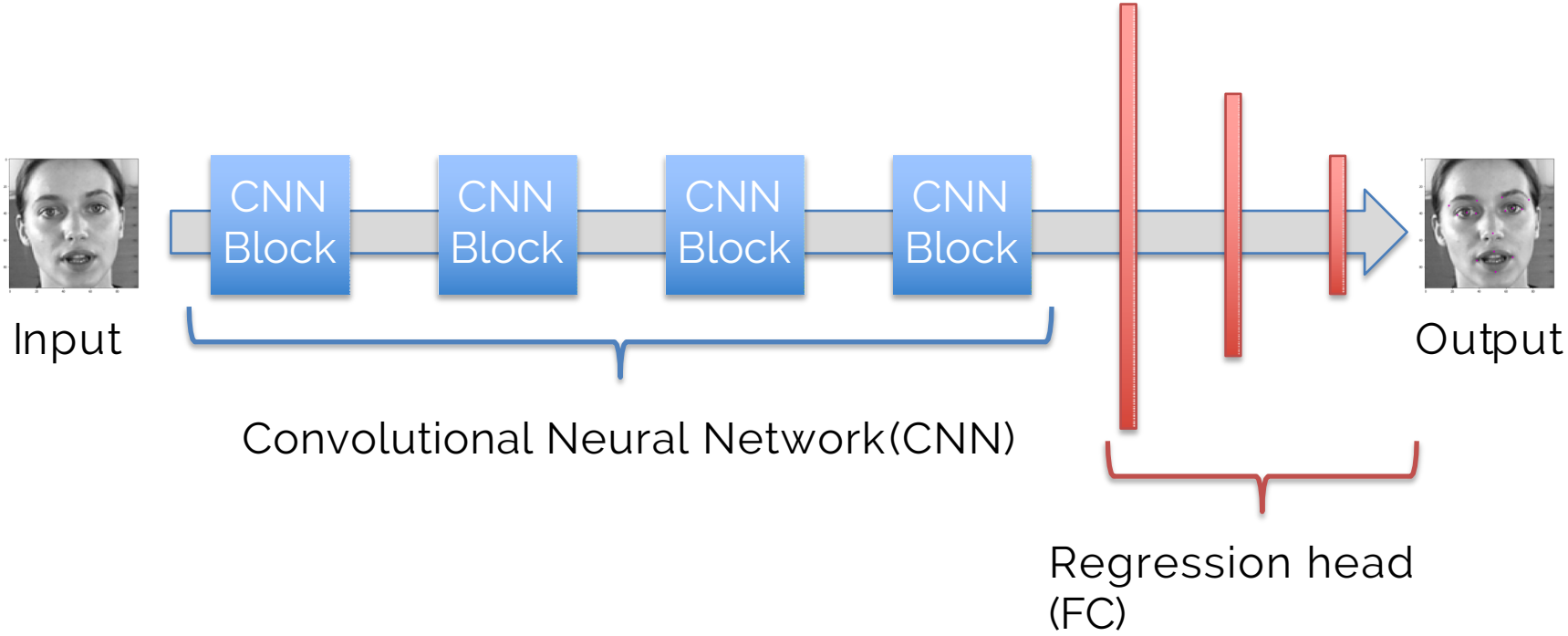
```
35      def __getitem__(self, idx):
36          sample = {'image': None, 'keypoints': None}
37          ###########################################################################
38          # TODO:                                                                   #
39          # Return the idx sample in the Dataset. A sample should be a              #
40          # dictionary where the key, value should be like                          #
41          #        {'image': image of shape [C, H, W],                              #
42          #         'keypoints': keypoints of shape [num_keypoints,                 #
43          ###########################################################################
44
45          from exercise_code.data_utils import get_keypoints
46          from exercise_code.data_utils import get_image
47
48          image = get_image(idx, self.key_pts_frame)
49          # expand channel
50          image = np.expand_dims(image, axis=0).astype(np.uint8)
51          keypoints = get_keypoints(idx, self.key_pts_frame).astype(np.float)
52
53          sample = {'image': image, 'keypoints': keypoints}
54
55          ###########################################################################
```

```
249     def get_image(idx, key_pts_frame):
250         image_string = key_pts_frame.loc[idx]['Image']
251         return string2image(image_string)
252
253
254     def get_keypoints(idx, key_pts_frame):
255         keypoint_cols = list(key_pts_frame.columns)[:-1]
256         return key_pts_frame.iloc[idx][keypoint_cols].values.reshape((15, 2))\
257             .astype(np.float32)
```

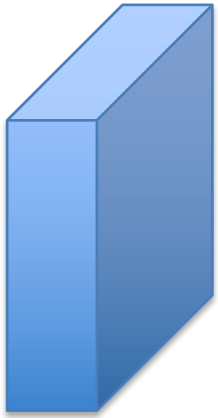
# Transforms

```
5   class Normalize(object):
6       """
7       Normalizes keypoints.
8       """
9       def __call__(self, sample):
10
11          image, key_pts = sample['image'], sample['keypoints']
12
13          ###############################################################
14          # TODO: Implemnet the Normalize function, where we normalize #
15          # the image from [0, 255] to [0,1] and keypoints from [0, 96]#
16          # to [-1, 1]                                                  #
17          ###############################################################
18
19          key_pts = (key_pts - 48.) / 48.
20          image = image / 255.0  *2 - 1
21
22          ###############################################################
23          # End of your code                                            #
24          ###############################################################
25          return {'image': image, 'keypoints': key_pts}
```

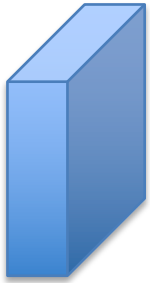
# Network Architecture



# CNN Block

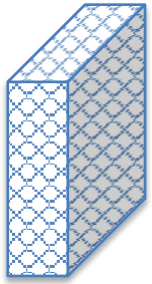


Conv Layer



Max Pooling

ReLU



(2dDropout)

# Input / Output size conv layer

$$W_O = (W_I - F + 2P) / S + 1$$

- $W_O$: Output Size
- $W_I$: Input Size
- $F$: Filter Size
- $P$: Zero Padding
- $S$: Stride

You should know!

Stride =1
Padding =1

# Network Architecture

$$W_O = (W_I - F + 2P)/S + 1$$

Output Size

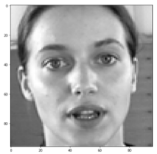
Input Size

Filter Size

Zero Padding

Stride

(1x 96 x 96) (32 x 48 x 48) (64 x 24 x 24) (128 x 12 x12) (256 x 6 x 6)



Input

CNN Block

CNN Block

CNN Block

CNN Block

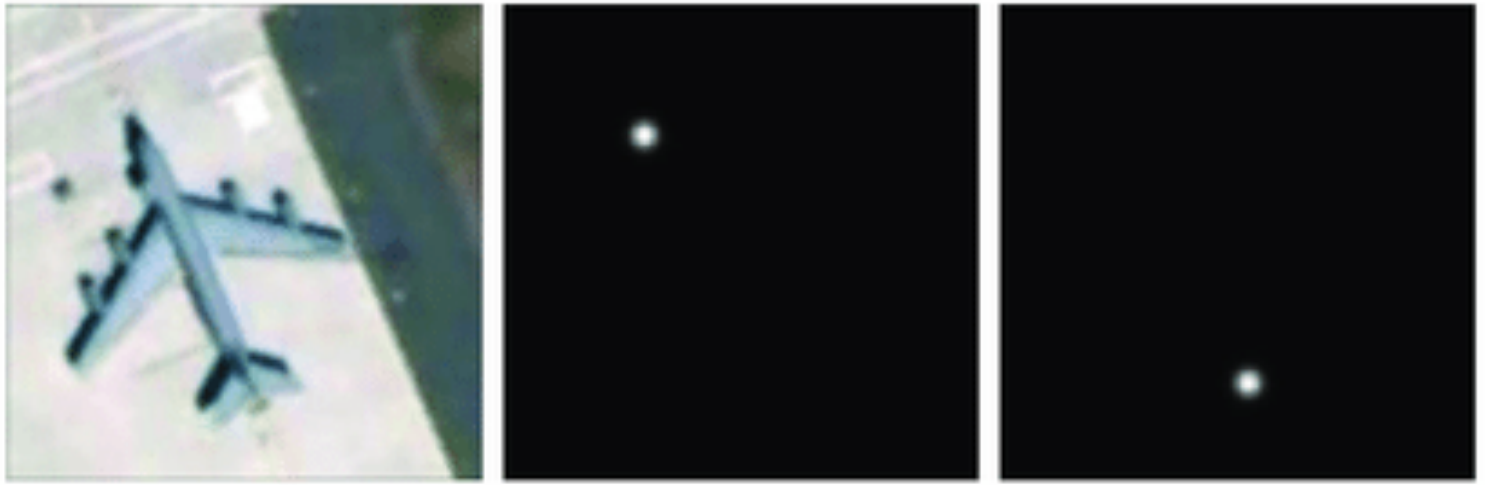
Convolutional Neural Network(CNN)

Stride =1
Padding =1

# Network Code

```
24
25          def conv_sandwich(inp, out, kernel_size, stride, pad):
26              return nn.Sequential(
27                  nn.Conv2d(inp, out, kernel_size, stride, pad),
28                  nn.MaxPool2d(2, 2),
29                  nn.ReLU()
30              )
31
32          layers = []
33          layers.append(conv_sandwich(1, 32, kernel_size=3, stride=1, pad=1))
34          layers.append(conv_sandwich(32, 64, kernel_size=3, stride=1, pad=1))
35          layers.append(conv_sandwich(64, 128, kernel_size=3, stride=1, pad=1))
36          layers.append(conv_sandwich(128, 256, kernel_size=3, stride=1, pad=1))
37          self.convs = nn.Sequential(*layers)
38
39          self.fc1 = nn.Sequential(nn.Linear(256 * 6 * 6, 256), nn.ReLU())
40          self.fc2 = nn.Sequential(nn.Linear(256, 30), nn.Tanh())
41
```

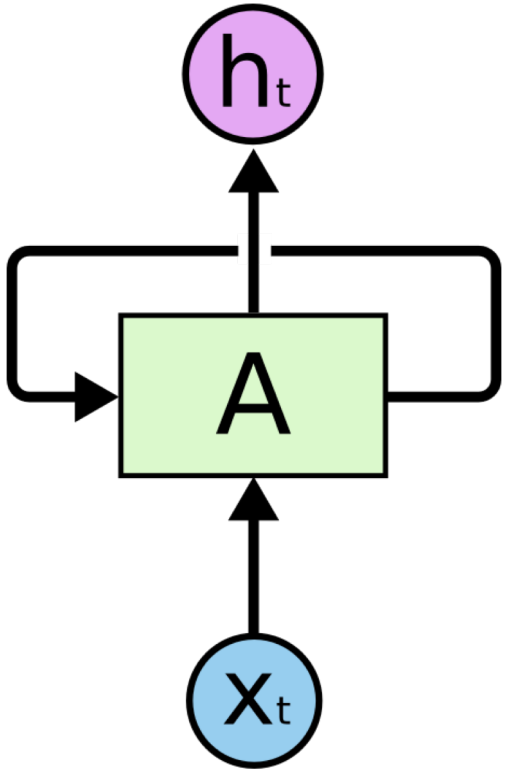
# Keypoints as Heatmaps



(a) (b) (c)

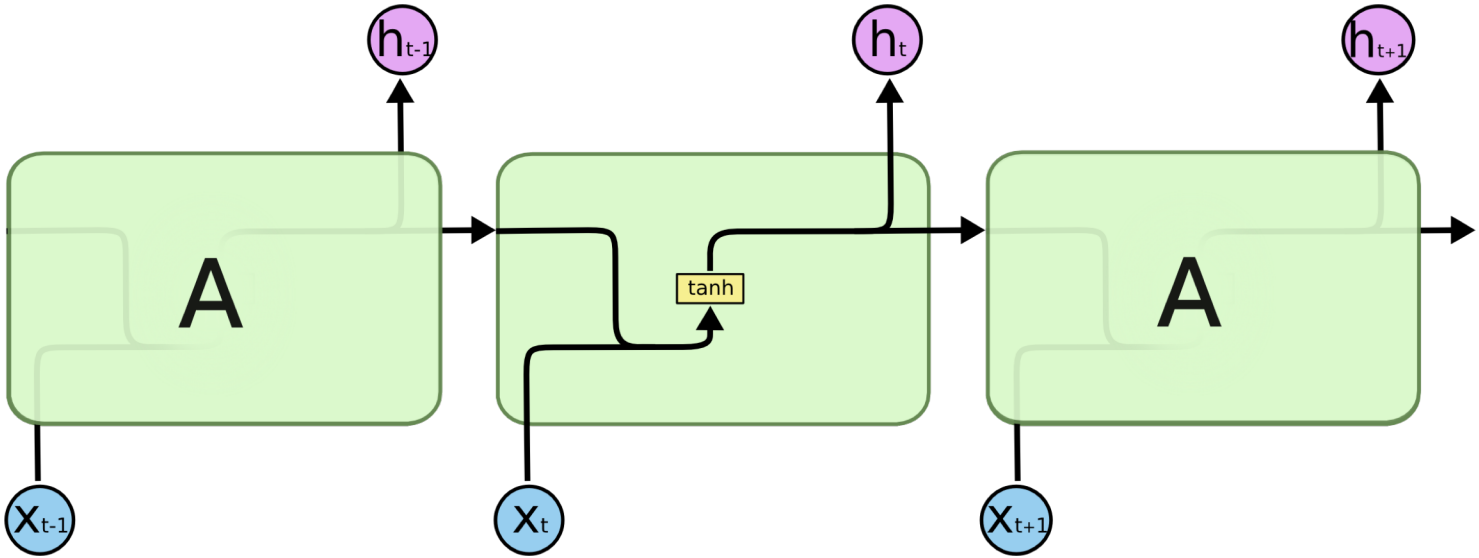
Loss in this situation: Softmax

# Recurrent Neural Nets



# Simple RNN

$$h_t = \tanh\left(W \cdot h_{t-1} + V \cdot x_t + b\right)$$



# Simple RNN

```
21          self.hidden_size = hidden_size
22          self.input_size = input_size
23
24          self.W_hh = nn.Linear(self.hidden_size, self.hidden_size, bias=True)
25          self.W_xh = nn.Linear(self.input_size, self.hidden_size, bias=True)
26          if activation == "tanh":
27              self.activation = nn.Tanh()
28          elif activation == "relu":
29              self.activation = nn.ReLU()
30
31          else:
32              raise ValueError(
33                  "Unrecognized activation. Allowed activations: tanh or relu")
```

# Simple RNN

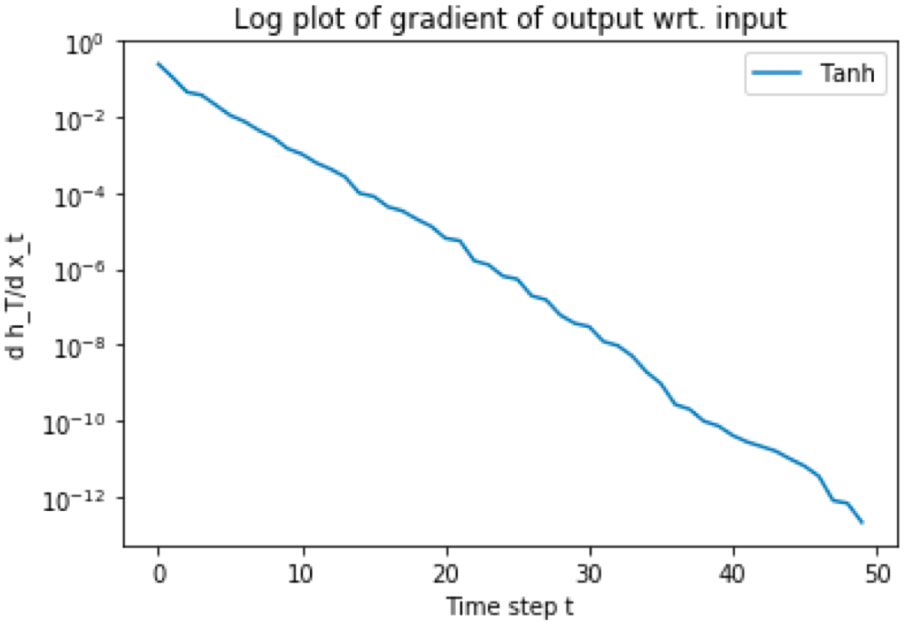
- Forward

```
55          if h is None:
56              h = torch.zeros((1, x.size(1), self.hidden_size)).float()
57
58          h_seq = []
59          for t in range(x.size(0)):
60              # update the hidden state
61
62              h = self.activation(self.W_hh(h) + self.W_xh(x[t]))
63              h_seq.append(h)
64          h_seq = torch.cat(h_seq, 0)
```

- Own RNN vs Pytorch:
  - Time Pytorch RNN 10000 runs: 2.796s
  - Time I2DL RNN 10000 run: 6.434s

# Simple RNN

- Vanishing gradient




$$h_t = \tanh\left(W \cdot h_{t-1} + V \cdot x_t + b\right)$$

$$\sigma(\cdot) = \tanh(\cdot) \qquad z_t = W \cdot h_{t-1} + V \cdot x_t$$

$$\frac{\partial h_t}{\partial x_\tau} = \frac{\partial \sigma(z_t)}{\partial z_t}\left(W \cdot \frac{\partial h_{t-1}}{\partial x_\tau} + V \cdot \delta_{t\tau}\right)$$

# Simple RNN

$$h_t = \tanh\left(W \cdot h_{t-1} + V \cdot x_t + b\right)$$
$$\sigma(\cdot) = \tanh(\cdot) \qquad z_t = W \cdot h_{t-1} + V \cdot x_t$$

- Calculate $\frac{\partial h_3}{\partial x_0}$ with $h_0 = 0$

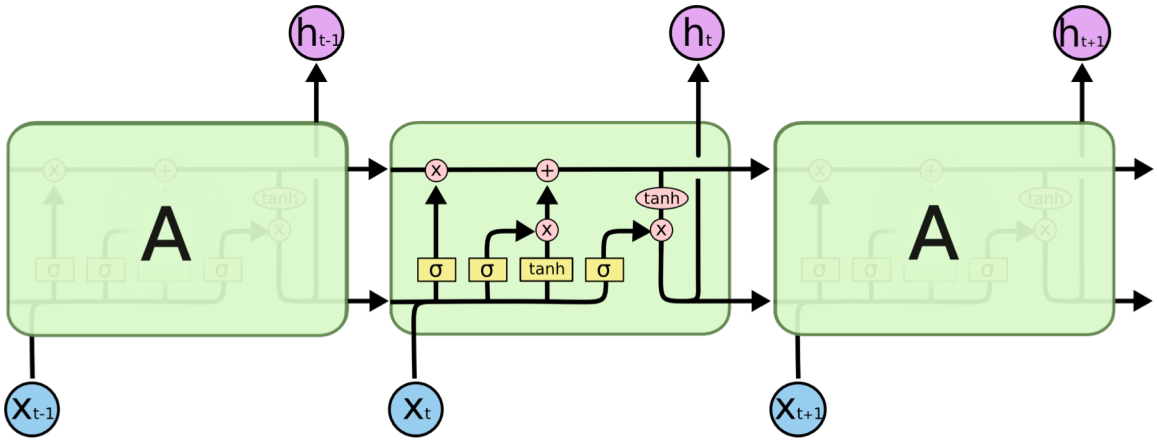
$$\frac{\partial h_t}{\partial x_\tau} = \frac{\partial \sigma(z_t)}{\partial z_t}\left(W \cdot \frac{\partial h_{t-1}}{\partial x_\tau} + V \cdot \delta_{t\tau}\right)$$

$$\frac{\partial h_3}{\partial x_1} = \frac{\partial \sigma(z_3)}{\partial z_3}\left(W \cdot \frac{\partial \sigma(z_2)}{\partial z_2}\left(W \cdot \frac{\partial \sigma(z_1)}{\partial z_1} \cdot V\right)\right)$$

-> Gradient depends on the norm of W

# LSTM

$$f_t = \sigma_g\left(W_f \cdot h_{t-1} + V_f \cdot x_t + b_f\right)$$
$$i_t = \sigma_g\left(W_i \cdot h_{t-1} + V_i \cdot x_t + b_i\right)$$
$$o_t = \sigma_g\left(W_o \cdot h_{t-1} + V_o \cdot x_t + b_o\right)$$
$$c_t = f_t \circ c_{t-1} + i_t \circ \tanh\left(W_c \cdot h_{t-1} + V_c \cdot x_t + b_c\right)$$
$$h_t = o_t \circ \tanh\left(c_t\right)$$



# LSTM

- Initialisation

```
80          ########################################################################
81
82          self.hidden_size = hidden_size
83          self.input_size = input_size
84
85          self.W_hh = nn.Linear(self.hidden_size, 4*self.hidden_size, bias=True)
86          self.W_xh = nn.Linear(self.input_size, 4*self.hidden_size, bias=True)
87
88          ########################################################################
```

# LSTM

- Forward

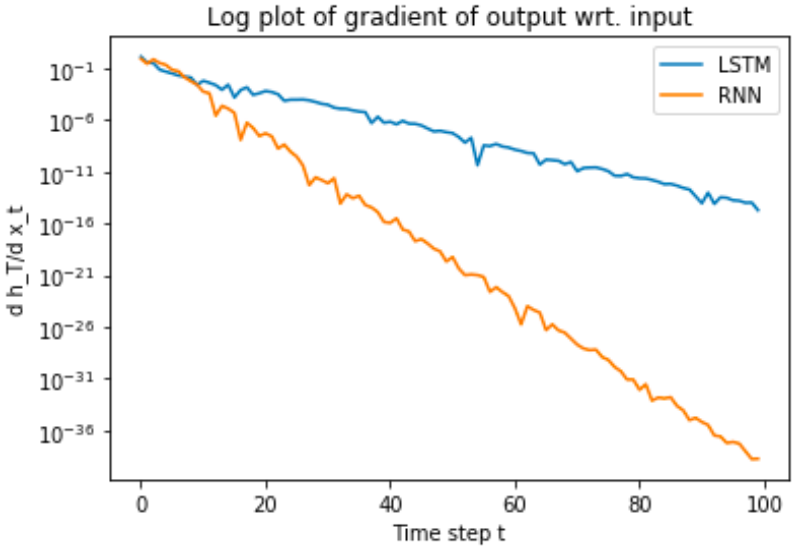
```
110         if h is None:
111             h = torch.zeros((1, x.size(1), self.hidden_size)).float()
112         if c is None:
113             c = torch.zeros((1, x.size(1), self.hidden_size)).float()
114         h_seq = []
115         for t in range(x.size(0)):
116             # update the hidden state
117             update_h = self.W_hh(h)
118             update_x = self.W_xh(x[t].unsqueeze(0))
119 
120             gates = (update_h[:, :, : 3 * self.hidden_size] +
121                      update_x[:, :, : 3 * self.hidden_size]).sigmoid()
122             fg = gates[:, :, :self.hidden_size]
123             ig = gates[:, :, self.hidden_size:2 * self.hidden_size]
124             og = gates[:, :, 2 * self.hidden_size:]
125             update = (update_h[:, :, 3 * self.hidden_size:] +
126                       update_x[:, :,  3 * self.hidden_size:]).tanh()
127             c = c * fg + update * ig
128             h = og * c.tanh()
129 
130             h_seq.append(h)
131         h_seq = torch.cat(h_seq, 0)
```

Own LSTM vs Pytorch:
Time Pytorch LSTM 10000 runs: **6.669s**
Time I2DL LSTM 10000 run: **16.756s**

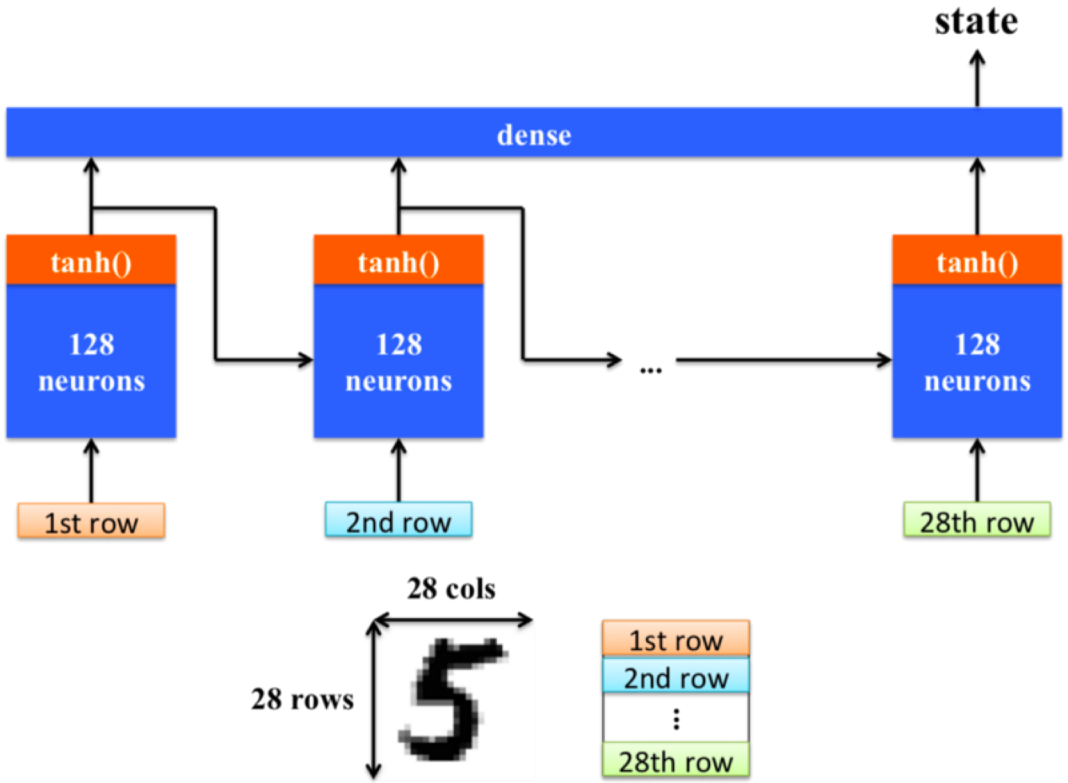
# LSTM

- LSTM "resolves" vanishing gradient problem

```
hidden_size=1
input_size= 1

time_steps=50
rnn=RNN(input_size, hidden_size)
X = torch.randn(time_steps, 1, input_size)
X.requires_grad=True
_,h=rnn(X)
h.requires_grad
h.sum().backward()
grad_rnn=X.grad.view(-1)

lstm=LSTM(input_size, hidden_size)
X = torch.randn(time_steps, 1, input_size)
X.requires_grad=True
_,(h, c)=lstm(X)
h.sum().backward()
grad_lstm=X.grad.view(-1)

plt.semilogy(np.flip(abs(grad_lstm.detach().cpu().numpy())) , label="LSTM")
plt.semilogy(np.flip(abs(grad_rnn.detach().cpu().numpy())), label="RNN")
plt.legend()
plt.xlabel("Time step t")
plt.ylabel("d h_T/d x_t")
plt.title("Log plot of gradient of output wrt. input")
plt.show()
```



# MNIST RNN Classifier



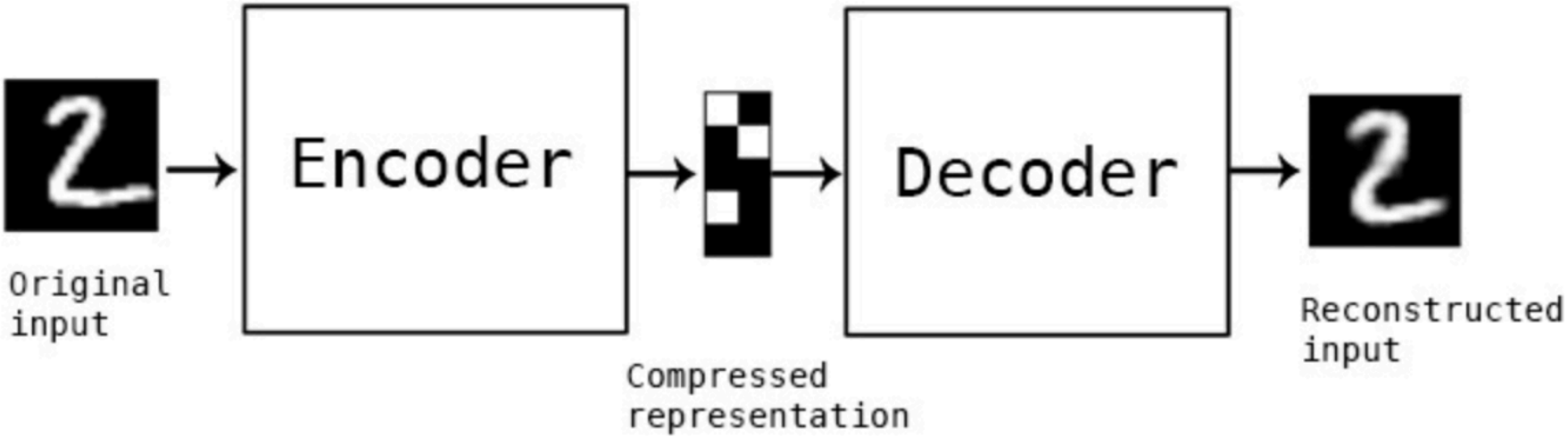
# MNIST RNN Classifier

```
class LSTM_Classifier(torch.nn.Module):
    def __init__(self, classes=10, input_size=28, hidden_size=128):
        super(LSTM_Classifier, self).__init__()
        ################################################################################
        #  TODO: Build a LSTM classifier                                               #
        ################################################################################

        self.LSTM = nn.LSTM(input_size=input_size, hidden_size=hidden_size)
        self.predictor = nn.Linear(hidden_size, classes)
        self.softmax = nn.Softmax(dim=2)
        """
        for m in self.RNN.modules():
            if isinstance(m, nn.Linear):
                nn.init.normal_(m.weight, mean=0, std=0.1)
                nn.init.constant_(m.bias, val=0)
        """


    def forward(self, x):
        _, (h, c) = self.LSTM(x)
        y = self.predictor(h)
        prediction = self.softmax(y)

        return prediction.squeeze(0)
```

# Segmentation

- Pixel-wise segmentation
- $Seg : \mathbb{R}^{n \times m \times 3} \rightarrow \mathbb{R}^{n \times m \times C}$

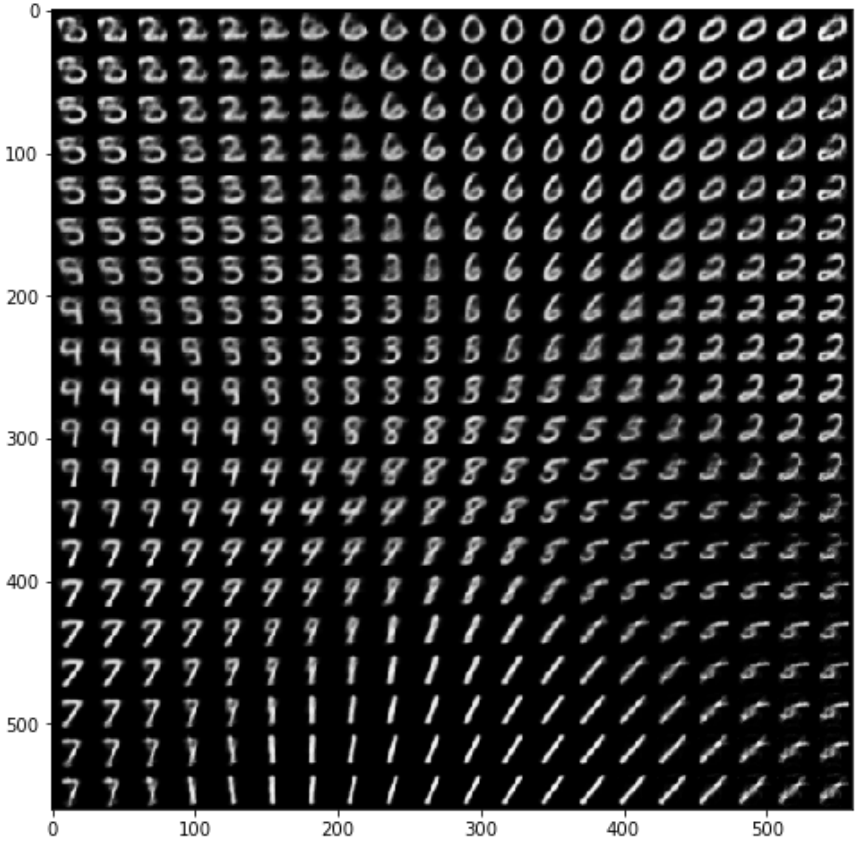
# Autoencoder architecture



# Possible: Simple Autoencoder

```
class Autoencoder(nn.Module):
    def __init__(self):
        super(Autoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Conv2d(1, 16, 3, stride=3, padding=1),  # size: 10, 10
            nn.ReLU(True),
            nn.MaxPool2d(2, stride=2),  # size: 5, 5
            nn.Conv2d(16, 8, 3, stride=2, padding=1),  # size: 3, 3
            nn.ReLU(True),
            nn.MaxPool2d(2, stride=1),  # size: 2, 2
        )
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(8, 16, 3, stride=2),  # size: 5, 5
            nn.ReLU(True),
            nn.ConvTranspose2d(16, 8, 5, stride=3, padding=1),  # size: 15, 15
            nn.ReLU(True),
            nn.ConvTranspose2d(8, 1, 2, stride=2, padding=1),  # size: 28, 28
            nn.Tanh()
        )
        self.bottleneck = nn.Sequential(
            nn.Linear(32, 32),
            nn.Tanh(),      # We use one tanh at the end so that we can sample a vector between
                            # -1 and 1 later on
        )
```

# Latent Space Interpolation (Variational Autoencoders)



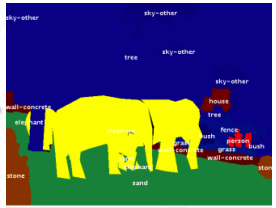
# Vanilla Segmentation architecture

Original Image

Encoder

Compressed representation

Decoder

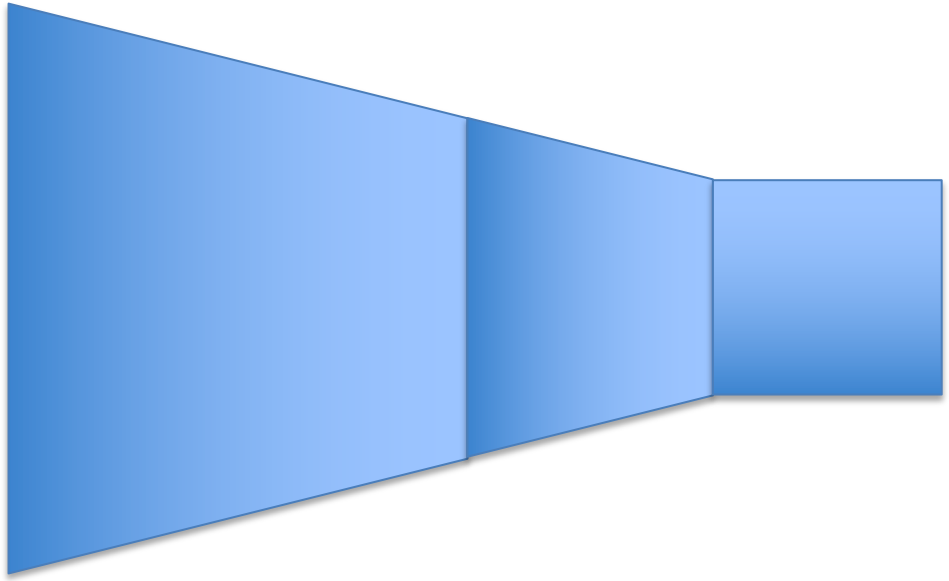


Segmentation map

- Encoder extracts features of images
- Image features can be independent of task

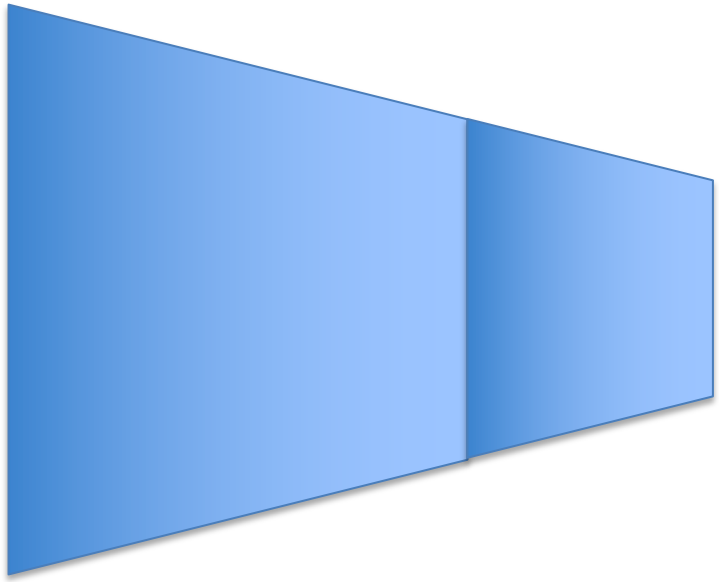
Idea: Transfer Learning

# Transfer Learning

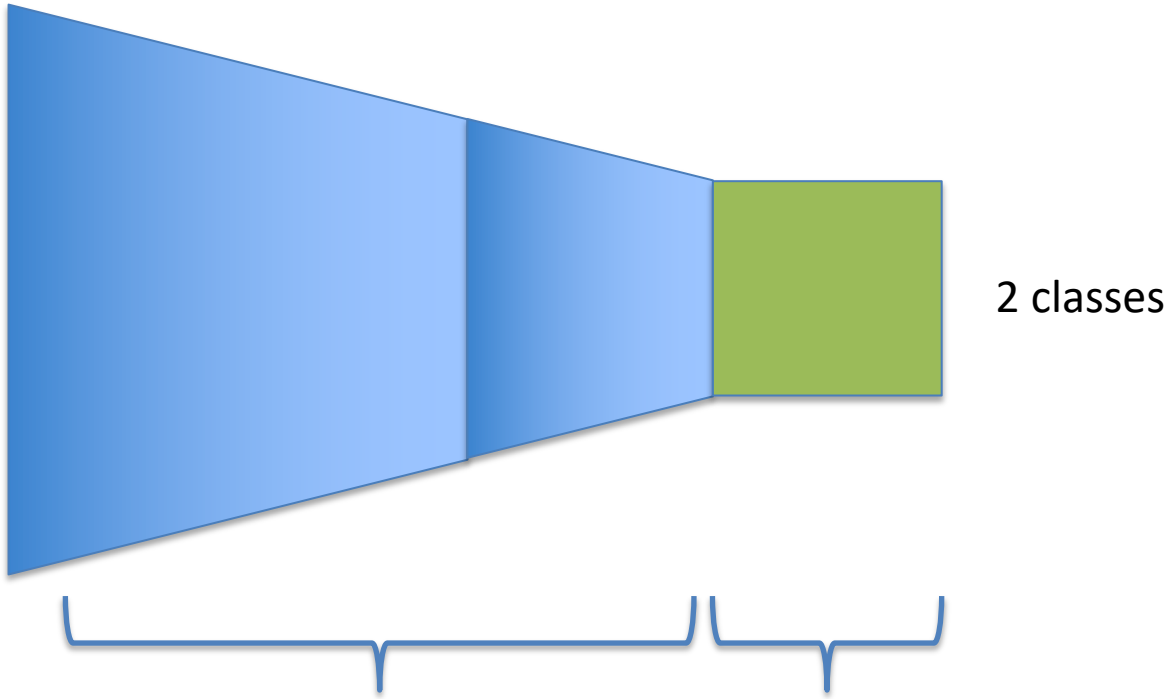
1000 classes

# Transfer Learning

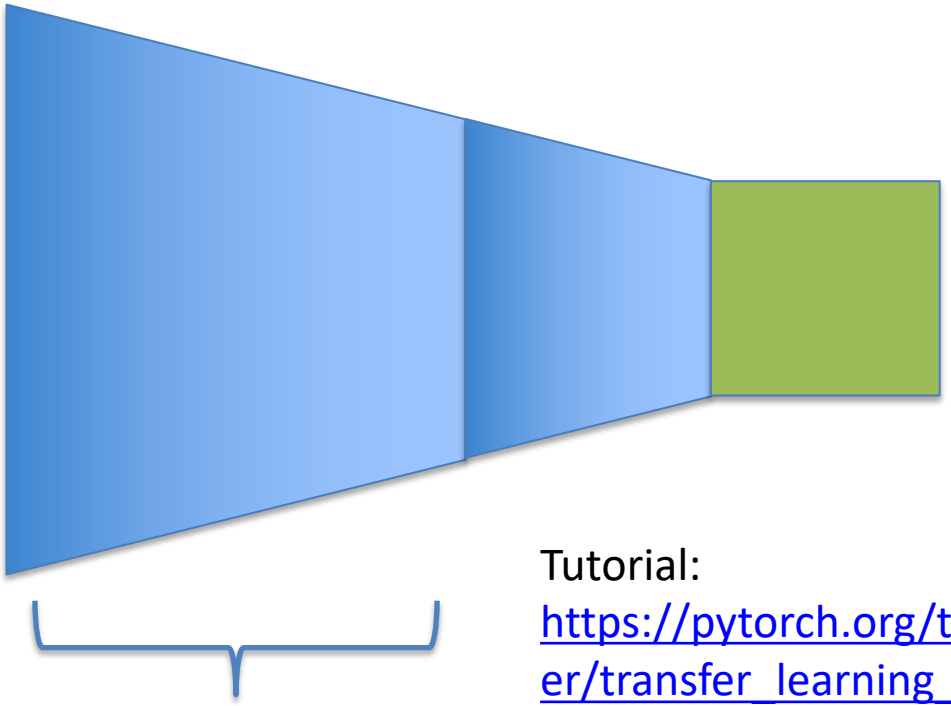


2 classes

# Transfer Learning: (Pre-)train



# Transfer Learning: Train




Tutorial:
https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html

# Transfer Learning

- Idea:
  - take network trained on a similar task with weights
  - Add/Remove layers to make it suitable for your task
  - Two options:
    - Define different learning rates for pretrained part and new parts
    - Freeze pretrained part for some epochs (recommended)
- Advantage:
  - Don't have to learn early convolutional weights who don't really care about the exact image contents
  - Able to keep filters in the latter part of the network for new task

# Segmentation Exercise

```
class SegmentationNN(nn.Module):

    def __init__(self, num_classes=23):
        super(SegmentationNN, self).__init__()

        ################################################################################
        #                                  YOUR CODE                                   #
        ################################################################################

        from torchvision import models
        self.features = models.alexnet(pretrained=True).features
        self.classifier = nn.Sequential(
            nn.Dropout(),
            nn.Conv2d(256, 4096, kernel_size=1, padding=0),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Conv2d(4096, 4096, kernel_size=1, padding=0),
            nn.ReLU(inplace=True),
            nn.Conv2d(4096, num_classes, kernel_size=1, padding=0),
            nn.Upsample(scale_factor=40),
            nn.Conv2d(num_classes, num_classes, kernel_size=3, padding=1),
        )
```

# Segmentation Exercise

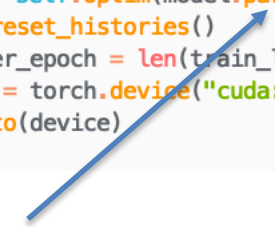
How to do it:

- Use a pretrained CNN (i.e. AlexNet) as feature extractor
- Add some convolutional layers to train decoder
- Upscale to output dimension
  - Here: simple upsampling

```
solver = Solver(optim_args={"lr": 1e-4, "weight_decay": 0.001},
                loss_func=torch.nn.CrossEntropyLoss(size_average=True, ignore_index=-1))
solver.train(model, train_loader, val_loader, log_nth=5, num_epochs=5)
```

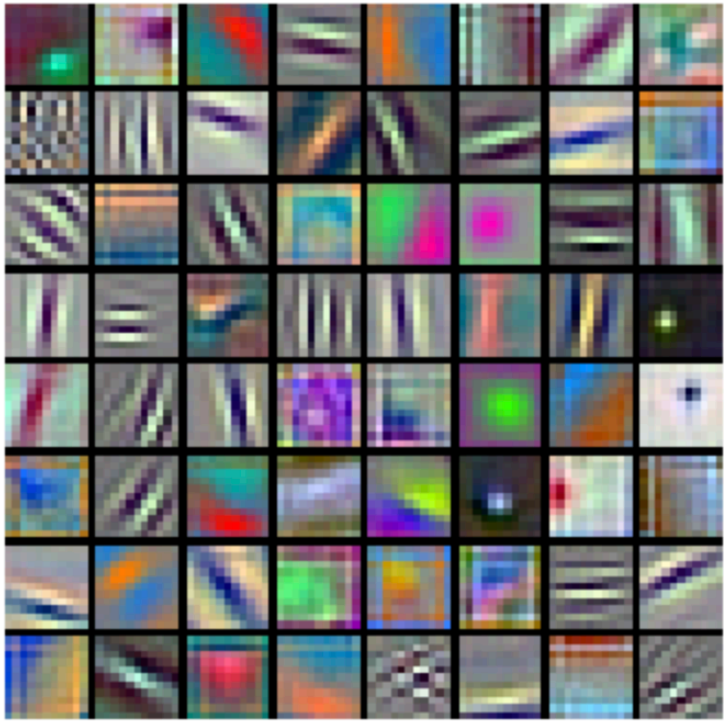
# How to freeze weights?

```
def train(self, model, train_loader, val_loader, num_epochs=10, log_nth=0):
    """
    Train a given model with the provided data.

    Inputs:
    - model: model object initialized from a torch.nn.Module
    - train_loader: train data in torch.utils.data.DataLoader
    - val_loader: val data in torch.utils.data.DataLoader
    - num_epochs: total number of training epochs
    - log_nth: log training accuracy and loss every nth iteration
    """
    optim = self.optim(model.parameters(), **self.optim_args)
    self._reset_histories()
    iter_per_epoch = len(train_loader)
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
    model.to(device)
```

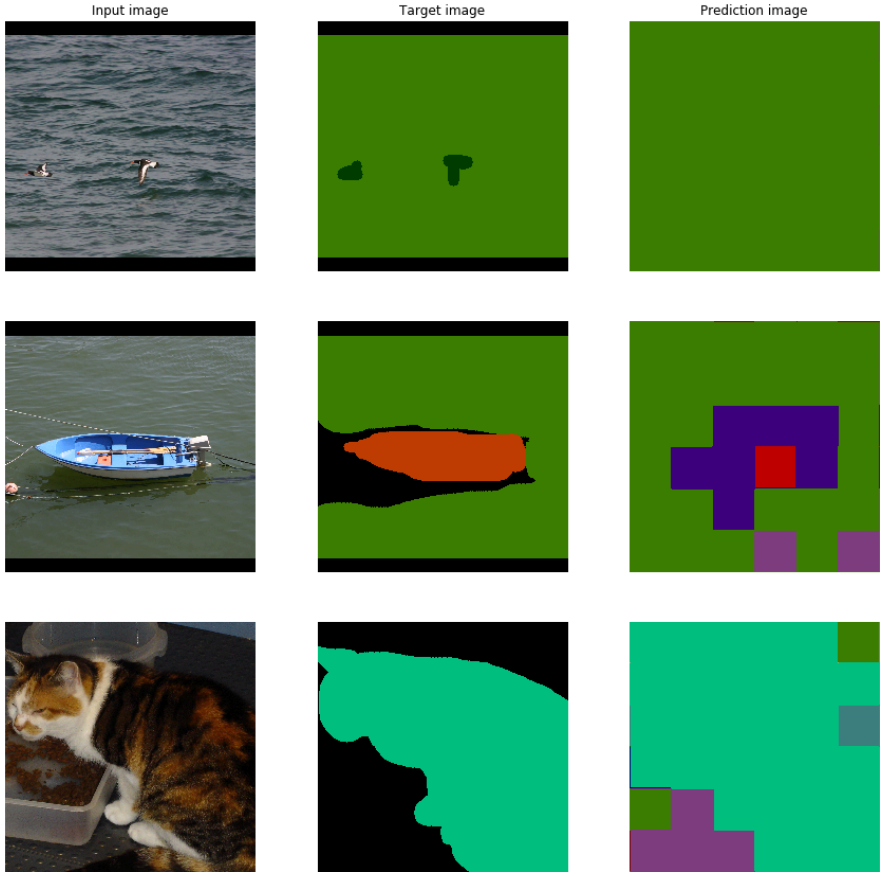


model.classifier.parameters()

# Visualize filter weights



# Outputs...



# Best performing student model

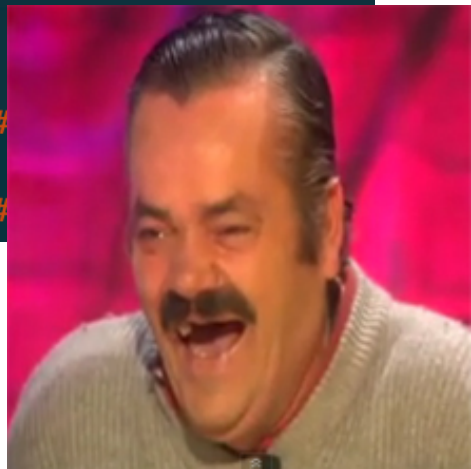
## Exercise Submission for I2DL

Submission 0 | Submission 1 | Submission 2 | Submission 3 | Submission 4 | **Submission 5** | Submission 6 | Submission 7 | Bonus

| Rank | User | Date submitted | Score | Pass |
|---|---|---|---|---|
| **Submission 5** | | | | |
| #1 | w0256 | 2020-01-14 07:42:59 | 93.37 | ✔ |
| #2 | w0220 | 2020-01-22 01:43:11 | 92.61 | ✔ |
| #3 | w0445 | 2019-12-26 16:17:56 | 92.49 | ✔ |

# What do they do better?

```
################################################################################
#                                YOUR CODE                                     #
################################################################################
from torchvision.models.segmentation.deeplabv3 import DeepLabHead
from torchvision import datasets, models, transforms
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

self.deeplab = models.segmentation.deeplabv3_resnet101(pretrained=True, progress=True)
# Change output size
self.deeplab.classifier = DeepLabHead(2048, num_classes)
self.deeplab.to(device)
################################################################################
#                            END OF YOUR CODE                                  #
################################################################################
```



# Practical (Pytorch) Tips

# Coding Project Outline

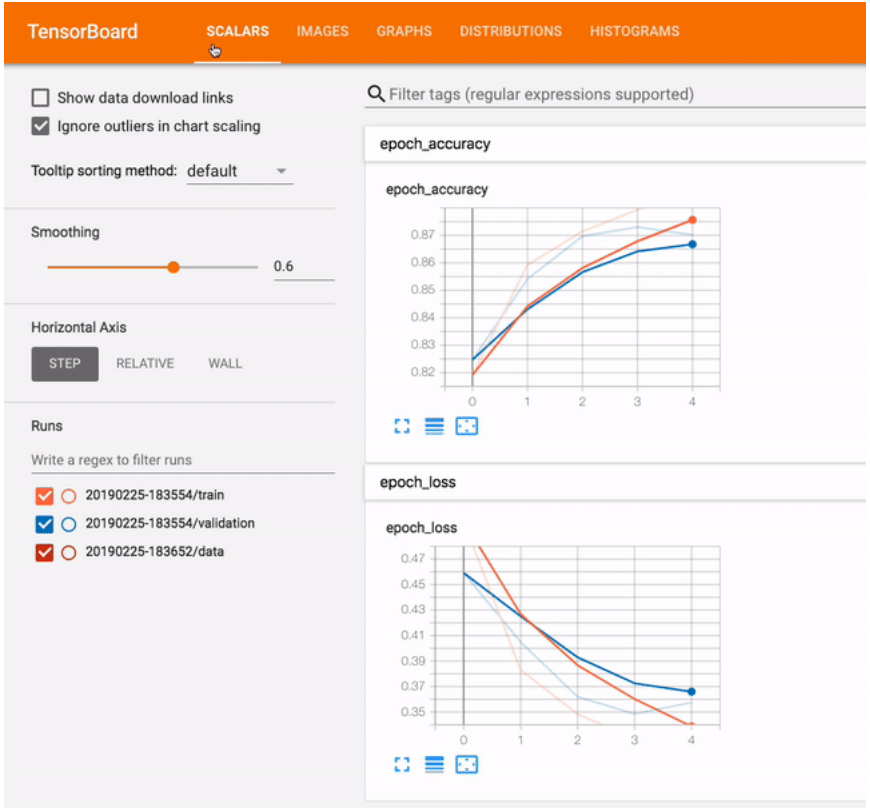
- Data
  - Write visualizer
  - Create data loader and unit test it
- Solver (main program)
  - Adopt from previous project or use existing libraries
  - Log all input variables
  - Always display all relevant graphs and numbers throughout the training process
    - Use existing tools like tensorboard

# Coding Project Outline

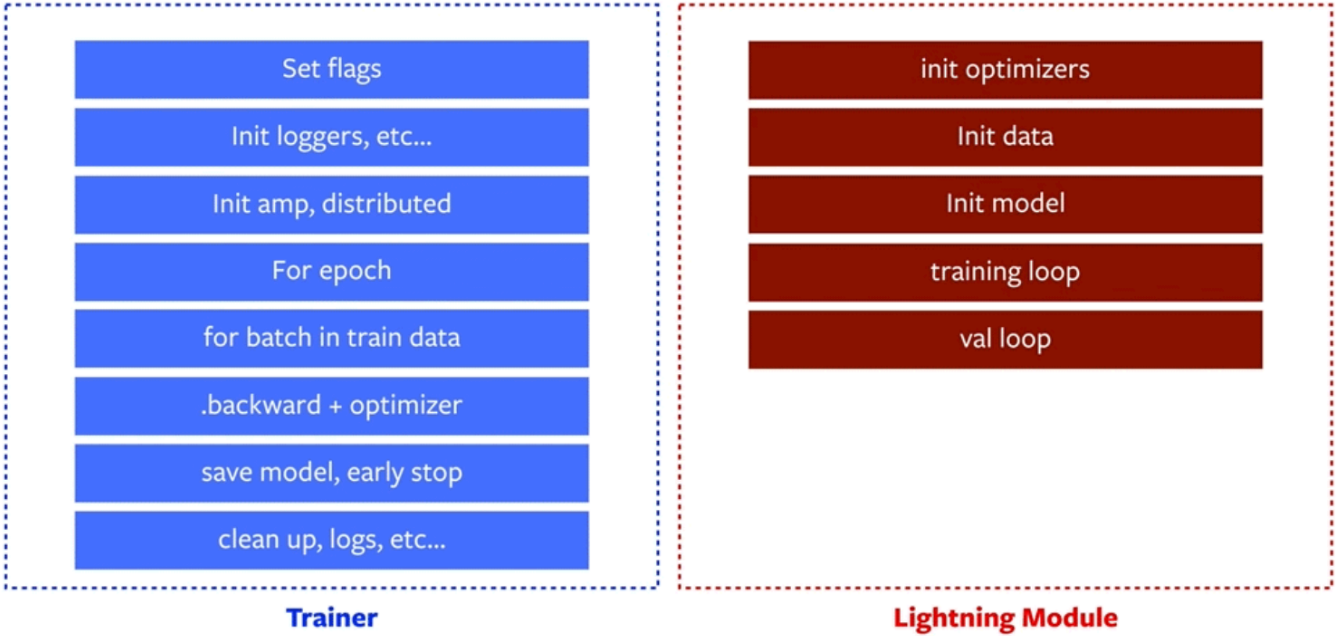
- Network
  - Start very simple (no pretrained model)
  - Overfit to a low number of examples to check dataloader (,solver) and network
  - Expand from there and always test multiple hyperparameters
- Other stuff
  - Implement data augmentation (transforms)

# How to visualize training?

- Use libraries like
  - Visdom
  - Tensorboard
    - Use pytorch wrappers like tensorboardX etc.



# Pre-existing solvers?




Link: https://github.com/PyTorchLightning/pytorch-lightning

# Exam

# Exam: Hard facts

- 90 minutes to achieve 90 points
- No calculator, laptop or script allowed
- 3 parts:
  - Multiple choice questions
  - Short questions
  - 4 long questions
- Covers lecture and exercise content
- No explicit coding but pseudo code possible

# Notes

- TUM Exam
  - There will be no official classroom review but everything will be done online
  - We will outline the necessary steps after the exam on moodle

- Read the instrutions on page 1 carefully!

# Notes

- Multiple Choice
  - Theme can change but will probably be as outlined in the mock exam

- Mock Exam
  - Solutions will be posted today
  - There can be errors in the solutions
  - all discussions on moodle

# Topics you should know!

List not exhaustive!

- **Lecrure 1: Intro**
  - Motivation and Applications
- **Lecture 2: Machine Learning Basics**
  - (Un-)Supervised Learning
  - (K-)Nearest Neighbors
  - Data Splitting + Cross Validation
  - (Linear/Logistic) Regression
- **Lecture 3: Intro to NNs**
  - Activation Functions
  - Loss Functions
  - Computational Graphs
- **Lecture 4: Backpropagation**
  - NNs as Computational Graphs
  - Backpropagation
  - Gradient Descent
- **Lecture 5: Optimization for NNs**
  - Stochastic Gradient Descent and deviations
  - Convergence of GD/ SGD
  - Newton's and other 2nd order methods
- **Lecture 6: Training NNs I**
  - Over- and Underfitting
  - Interpretation of train/ val curves
- **Lecture 7: Training NNs II**
  - Activation Functions and Loss Functions
  - Weight Initialization
- **Lecture 8: Advanced Regularization techinques**
  - Data Augmentation
  - Regularization (Normalization, Dropout, …)
- **Lecture 9: Intro to CNNs**
  - Convolutions (Forward passes, Dimensions)
  - Pooling
- **Lecture 10: CNN Architectures**
  - Receptive Field
  - Different Architectures (LeNet, AlexNet, VGGNet, ResNet, Inception Layer, EfficientNet, R-CNNs, FCNs, U-Net)
- **Lecture 11: RNNs**
  - Transfer Learning
  - RNNs & LSTMs
    - Dependencies and Vanishing Gradient
    - Shortcomings of RNNs
    - Improvements via LSTM cells
- **Lecture 12: Autoencoder and GANs**
  - Encoder – Decoder Architecture with Use-Cases
  - Variational Autoencoder
  - GANs with applications

# Organization

# Organization

- Office hours
  - Will continue until the exam
- Next lecture:
  - Guest lecture with Timo Aila

Questions?

Good luck for the exam☺

19.02.20 – 13:30h