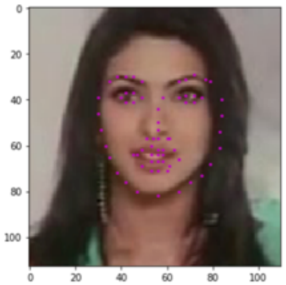
Introduction to Deep Learning (I2DL)

Exercise 9: Facial Keypoint Detection

Overview

- Exercise 8 Recap
  - Case study of two submitted solutions
- Convolutional Layers
  - Implementations
  - Changes to Dropout & Batchnorm
- Submission: Facial Keypoint Detection
  - Deadline: July 03, 2020 23:59



Case Study: Leaderboard

Leaderboard: Submission 8

Rank	User	Score	Pass
#1	s0499	67.09	✔
#2	s0770	66.18	✔
#3	s0631	60.79	✔
#4	s0697	60.19	✔
#5	s0641	58.77	✔
#6	s0446	58.57	✔
#7	s0385	57.63	✔
#8	s0322	57.40	✔
#9	s0332	57.23	✔
#10	s0739	57.00	✔

Case Study #1: Model

```python
######################################################################
# TODO: Initialize your model!                                       #
######################################################################

def init_weights(m):
    if type(m) == nn.Linear:
        torch.nn.init.xavier_normal_(m.weight)
        m.bias.data.fill_(0.01)

inputs = input_size
var_dropout = self.hparams["dropout"]
var_testnum = self.hparams["width"]

self.model = nn.Sequential(
    nn.Linear(inputs, var_testnum),
    nn.BatchNorm1d(var_testnum),
    nn.ReLU(),
    nn.Dropout(var_dropout),
    nn.Linear(var_testnum, var_testnum//2),
    nn.BatchNorm1d(var_testnum//2),
    nn.ReLU(),
    nn.Dropout(var_dropout),
    nn.Linear(var_testnum//2, var_testnum//4),
    nn.BatchNorm1d(var_testnum//4),
    nn.ReLU(),
    nn.Dropout(var_dropout),
    nn.Linear(var_testnum//4, var_testnum//8),
    nn.BatchNorm1d(var_testnum//8),
    nn.ReLU(),
    nn.Dropout(var_dropout),
    nn.Linear(var_testnum//8, var_testnum//4),
    nn.BatchNorm1d(var_testnum//4),
    nn.ReLU(),
    nn.Dropout(var_dropout),
    nn.Linear(var_testnum//4, var_testnum//2),
    nn.BatchNorm1d(var_testnum//2),
    nn.ReLU(),
    nn.Dropout(var_dropout),
    nn.Linear(var_testnum//2, var_testnum),
    nn.ReLU(),
    nn.Linear(var_testnum, 10)
)

self.model.apply(init_weights)
```

Pytorch Default Weight Initialization

CLASS `torch.nn.Linear(in_features, out_features, bias=True)` [SOURCE]

Applies a linear transformation to the incoming data: $y = xA^T + b$

Parameters

- **in_features** – size of each input sample
- **out_features** – size of each output sample
- **bias** – If set to `False`, the layer will not learn an additive bias. Default: `True`

Shape:

- Input: $(N, *, H_{in})$ where $*$ means any number of additional dimensions and $H_{in} = \text{in\_features}$
- Output: $(N, *, H_{out})$ where all but the last dimension are the same shape as the input and $H_{out} = \text{out\_features}$.

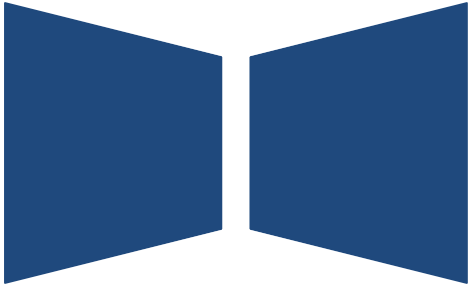
Variables

- **~Linear.weight** – the learnable weights of the module of shape $(\text{out\_features}, \text{in\_features})$. The values are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$, where $k = \frac{1}{\text{in\_features}}$
- **~Linear.bias** – the learnable bias of the module of shape (out_features). If `bias` is `True`, the values are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{\text{in\_features}}$

He Init
in comparison

$$Var(w_i) = \frac{2}{fan\_in}$$

Case Study #1: Model



```python
###########################################################################
# TODO: Initialize your model!                                            #
###########################################################################

def init_weights(m):
    if type(m) == nn.Linear:
        torch.nn.init.xavier_normal_(m.weight)
        m.bias.data.fill_(0.01)

inputs = input_size
var_dropout = self.hparams["dropout"]
var_testnum = self.hparams["width"]

self.model = nn.Sequential(
    nn.Linear(inputs, var_testnum),
    nn.BatchNorm1d(var_testnum),
    nn.ReLU(),
    nn.Dropout(var_dropout),
    nn.Linear(var_testnum, var_testnum//2),
    nn.BatchNorm1d(var_testnum//2),
    nn.ReLU(),
    nn.Dropout(var_dropout),
    nn.Linear(var_testnum//2, var_testnum//4),
    nn.BatchNorm1d(var_testnum//4),
    nn.ReLU(),
    nn.Dropout(var_dropout),
    nn.Linear(var_testnum//4, var_testnum//8),
    nn.BatchNorm1d(var_testnum//8),
    nn.ReLU(),
    nn.Dropout(var_dropout),
    nn.Linear(var_testnum//8, var_testnum//4),
    nn.BatchNorm1d(var_testnum//4),
    nn.ReLU(),
    nn.Dropout(var_dropout),
    nn.Linear(var_testnum//4, var_testnum//2),
    nn.BatchNorm1d(var_testnum//2),
    nn.ReLU(),
    nn.Dropout(var_dropout),
    nn.Linear(var_testnum//2, var_testnum),
    nn.ReLU(),
    nn.Linear(var_testnum, 10)
)

self.model.apply(init_weights)
```

Case Study #1: Transforms

```python
#########################################################################
# TODO: Define your transforms (convert to tensors, normalize).        #
# If you want, you can also perform data augmentation!                 #
#########################################################################
mean=[0.485, 0.456, 0.406]
std=[0.229, 0.224, 0.225]

my_transform = transforms.Compose([
    transforms.RandomCrop(size=32, padding=4),
    # transforms.ColorJitter(brightness=0.3, contrast=0.6, saturation=0.5)
    # transforms.RandomAffine(degrees=(10,90), translate=(0.1,0.3)),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize(mean, std, inplace=False)
])
```

Case Study #1: Tuning with Optuna

```python
def objective(trial):
    # as explained above, we'll use this callback to collect the validation accuracies
    metrics_callback = MetricsCallback()

    # create a trainer
    trainer = pl.Trainer(
        logger=True,                                                    # deactivate PL logging
        max_epochs=1000,                                                # epochs
        gpus=1 if torch.cuda.is_available() else None,                  # #gpus
        callbacks=[metrics_callback],                                   # save latest accuracy
        early_stop_callback=PyTorchLightningPruningCallback(trial, monitor='val_loss'), # early stopping
        Profiler = True
    )

    # here we sample the hyper params, similar as in our old random search
    trial_hparams = {
                    "dropout": trial.suggest_loguniform("dropout", 0.2, 0.4),
                    "width": trial.suggest_int("width", 800, 1024),
                    "batch_size": 512,
                    "learning_rate": trial.suggest_loguniform("lr", 1e-4, 4e-4),
                    }

    # create model from these hyper params and train it
    model = MyPytorchModel(trial_hparams)
    model.prepare_data()
    trainer.fit(model)

    # save model
    save_model(model, '{}.p'.format(trial.number), "checkpoints")

    # return validation accuracy from latest model, as that's what we want to minimize by our hyper param search
    return metrics_callback.metrics[-1]["val_acc"]
```

Case Study #1: Hyperparameters

```python
from exercise_code.MyPytorchModel import MyPytorchModel

hparams = {}
########################################################################
# TODO: Define your hyper parameters here!                             #
########################################################################

hparams = {
    "batch_size": 512,
    "learning_rate": 0.00023581258276405585,
    "dropout": 0.2691774214827215,
    "width": 984
}

########################################################################
#                           END OF YOUR CODE                           #
########################################################################
model = MyPytorchModel(hparams)
model = model.to(device)
model.prepare_data()
```

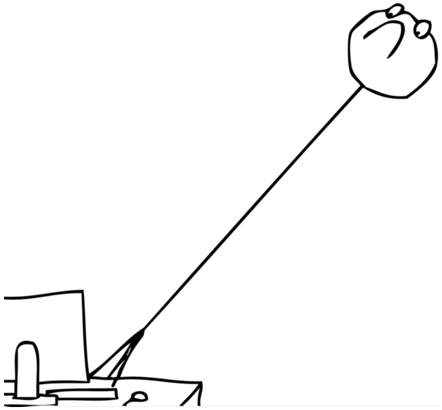
Case Study #2: Model

```python
self.model = nn.Sequential(
    nn.Linear(input_size, self.hparams["n_hidden"]),
    nn.BatchNorm1d(self.hparams["n_hidden"]),
    # nn.LeakyReLU(),
    nn.PReLU(self.hparams["n_hidden"]),
    nn.Dropout(0.5),

    nn.Linear(self.hparams["n_hidden"], self.hparams["n_hidden"]),
    nn.BatchNorm1d(self.hparams["n_hidden"]),
    # nn.LeakyReLU(),
    nn.PReLU(self.hparams["n_hidden"]),
    nn.Dropout(0.5),

    nn.Linear(self.hparams["n_hidden"], self.hparams["n_hidden"]),
    nn.BatchNorm1d(self.hparams["n_hidden"]),
    # nn.LeakyReLU(),
    nn.PReLU(self.hparams["n_hidden"]),
    nn.Dropout(0.5),

    nn.Linear(self.hparams["n_hidden"], num_classes)
)
```

Same number of hidden nodes for each layer
-> something to tune

Case Study #2: Transforms

```python
mean=[0.485, 0.456, 0.406]
std=[0.229, 0.224, 0.225]

# 1: Nearest; 2: Bilinear (lower quality, faster); 3: Bicubic (best quality, too slow).
# https://pytorch.org/docs/stable/torchvision/transforms.html
if self.hparams["augment"] == 0:
    print("Disabled data augmentation.")
    my_transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize(mean, std, inplace=False),
        # transforms.ToPILImage()
    ])
elif self.hparams["augment"] == 1:
    print("Enabling random transforms.")
    transform_list = [
        transforms.RandomResizedCrop(size=(32,32), scale=(0.75, 1.0), ratio=(0.8, 1.25), interpolation=3),
        transforms.RandomRotation(degrees=(-20,20), resample=3),
        transforms.ColorJitter(brightness=0.4, contrast=0.4, saturation=0.3, hue=0.1),
        transforms.RandomHorizontalFlip(p=0.9),
        transforms.RandomHorizontalFlip(p=0.01),
        transforms.RandomHorizontalFlip(p=0.01),
        # transforms.RandomHorizontalFlip(p=0.01),
        # transforms.RandomHorizontalFlip(p=0.01),
    ]
    my_transform = transforms.Compose([
        transforms.RandomChoice(transform_list),
        transforms.ToTensor(),
        transforms.Normalize(mean, std, inplace=False),
        # transforms.ToPILImage()
    ])
```



```python
elif self.hparams["augment"] == 2:
    print("Enabling full set of transforms.")
    my_transform = transforms.Compose([
        transforms.RandomResizedCrop(size=(32,32), scale=(0.75, 1.0), ratio=(0.8, 1.25), interpolation=3),
        transforms.RandomRotation(degrees=(-15,15), resample=3),
        transforms.ColorJitter(brightness=0.4, contrast=0.4, saturation=0.3, hue=0.1),
        transforms.RandomHorizontalFlip(p=0.2),
        transforms.ToTensor(),
        transforms.Normalize(mean, std, inplace=False),
        # transforms.ToPILImage()
    ])
else:
    print("Disabled data augmentation.")
    my_transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize(mean, std, inplace=False),
        # transforms.ToPILImage()
    ])
```

Case Study #2: Parameters

```python
from exercise_code.MyPytorchModel import MyPytorchModel

hparams = {}
########################################################################
# TODO: Define your hyper parameters here!                             #
########################################################################

hparams = {
    "batch_size": 128,
    "n_hidden": 64,
    "n_layer": 2,
    "weight_decay": 7e-7,
    # "learning_rate": 8e-4
}

########################################################################
#                           END OF YOUR CODE                           #
########################################################################
model = MyPytorchModel(hparams)
model.prepare_data()
```

Case Study #2: "Tuning"

```
#!pip install optuna
import optuna
from optuna.integration import PyTorchLightningPruningCallback
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
<ipython-input-2-f4c9d33445c9> in <module>
      1 #!pip install optuna
----> 2 import optuna
      3 from optuna.integration import PyTorchLightningPruningCallback

ModuleNotFoundError: No module named 'optuna'
```

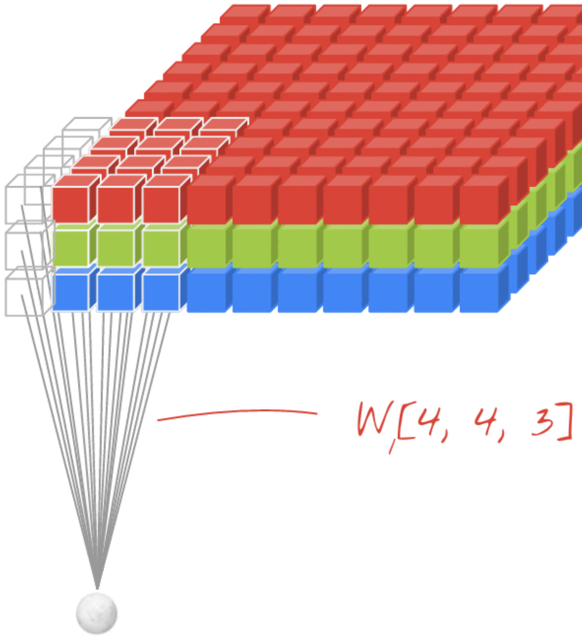
Moral:
use whatever method you prefer and continue to practice!

Convolutions vs Fully-Connected

- In contrast to fully-connected layers, we want to restrict the degrees of freedom
  - FC is somewhat brute force
  - Convolutions are **structured**

- Sliding window to with the same filter parameters to extract image features
  - Concept of weight sharing
  - Extract same features independent of location

Convolutions vs Fully-Connected



Source: Martin Görner

Weight shape for FC:
(N, I, O)

Weight shape for Conv:
(N, I, W, W, O)

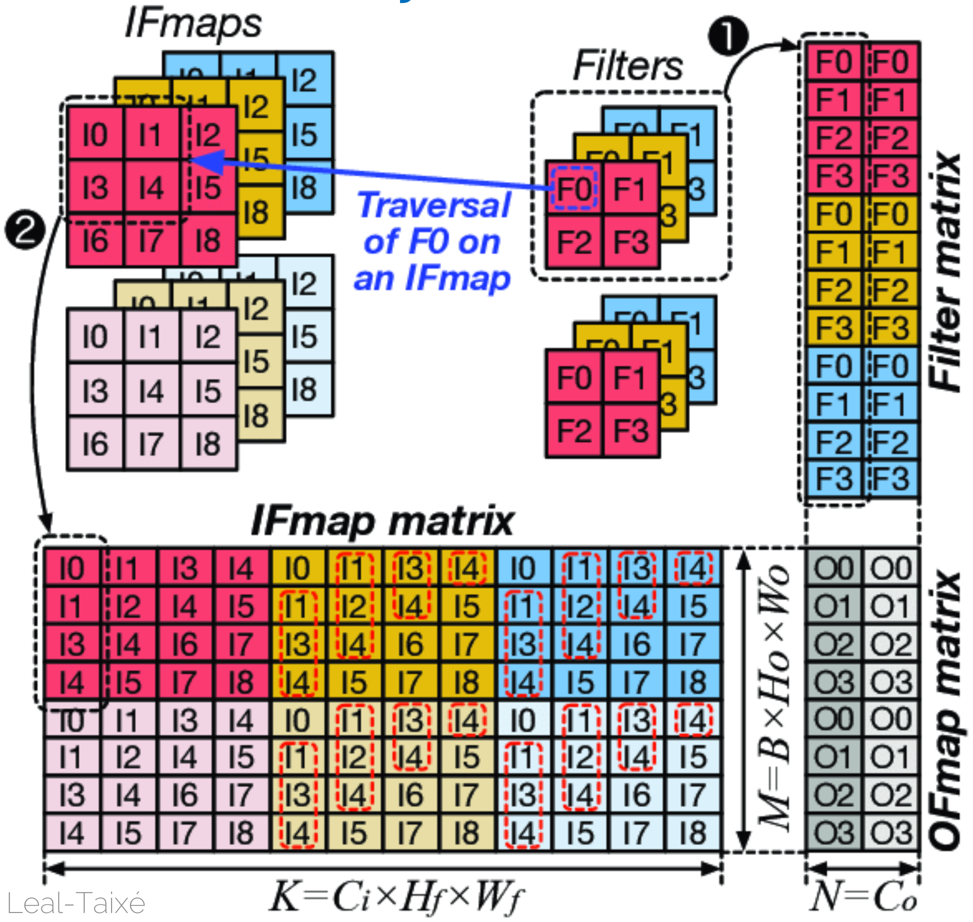
where
N: num samples in batch
I: num input features/channels
O: num output features/channels
W: Width of the conv kernel

Conv Layer - Naïve Implementation

```
N, C, H, W = x.shape
F, C, HH, WW = w.shape

pad = conv_param['pad']
stride = conv_param['stride']
x_pad = np.lib.pad(x,
                   ((0, 0), (0, 0), (pad, pad), (pad, pad)),
                   'constant',
                   constant_values=(0, 0))

H_convs = 1 + (H + 2 * pad - HH) // stride
W_convs = 1 + (W + 2 * pad - WW) // stride
out = np.zeros((N, F, H_convs, W_convs))
for img in range(N):
    for f in range(F):
        # The bias has to applied only once for each filter.
        out[img, f] += b[f]
        for i in range(H_convs):
            for j in range(W_convs):
                for channel in range(C):
                    # Add up all channels for a filter;
                    x_pad_slice = x_pad[img,
                                        channel,
                                        (i * stride):(i * stride + HH),
                                        (j * stride):(j * stride + WW)]
                    out[img, f, i,
                        j] += np.sum(x_pad_slice * w[f, channel, :, :])
```

```
# x = Input data of shape (N, C, H, W)
# w = Filter weights of shape (F, C, HH, WW)
# b = Biases, of shape (F,)
# out = the result of the convolution operation of shape
# (N, F, H', W'), where:
# H' = 1 + (H + 2 * pad - HH) / stride
# W' = 1 + (W + 2 * pad - WW) / stride
```

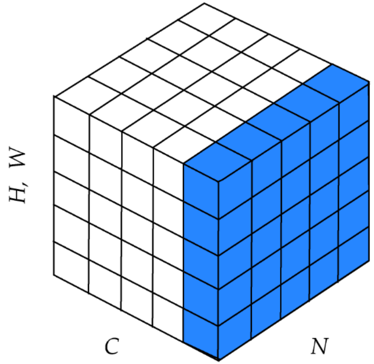
Conv Layer – Im2Col



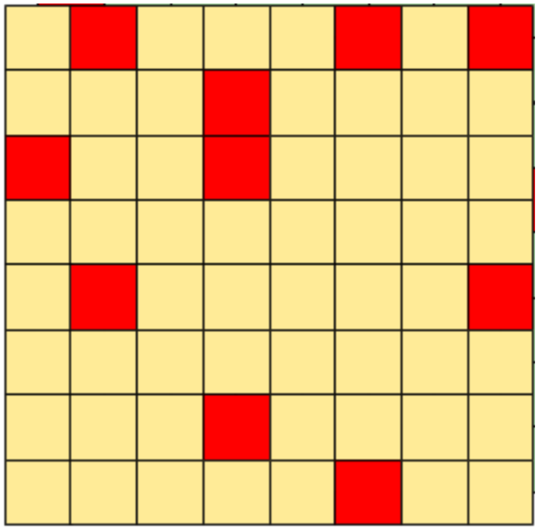
Spatial batchnorm

- Batchnorm for feedforward neural networks
  - Input size (N, D)
  - Compute minibatch mean and variance across N (i.e. mean.shape = (D,), variance.shape = (D,))

- Batchnorm for convolutional neural networks (spatial batchnorm)
  - Input size (N, C, W, H)
  - Compute minibatch mean and variance across N, W, H (i.e. mean.shape = (C,), variance.shape = (C,))

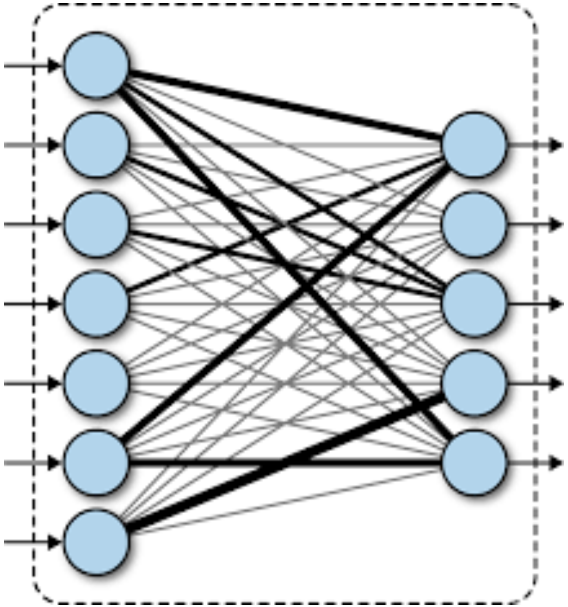


Dropout for Fully Connected

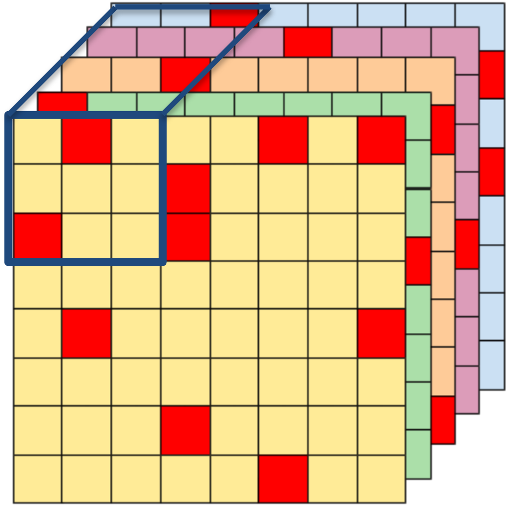


standard dropout: randomly drop units from the feature channels

 dropped unit



Dropout for convolutional layers



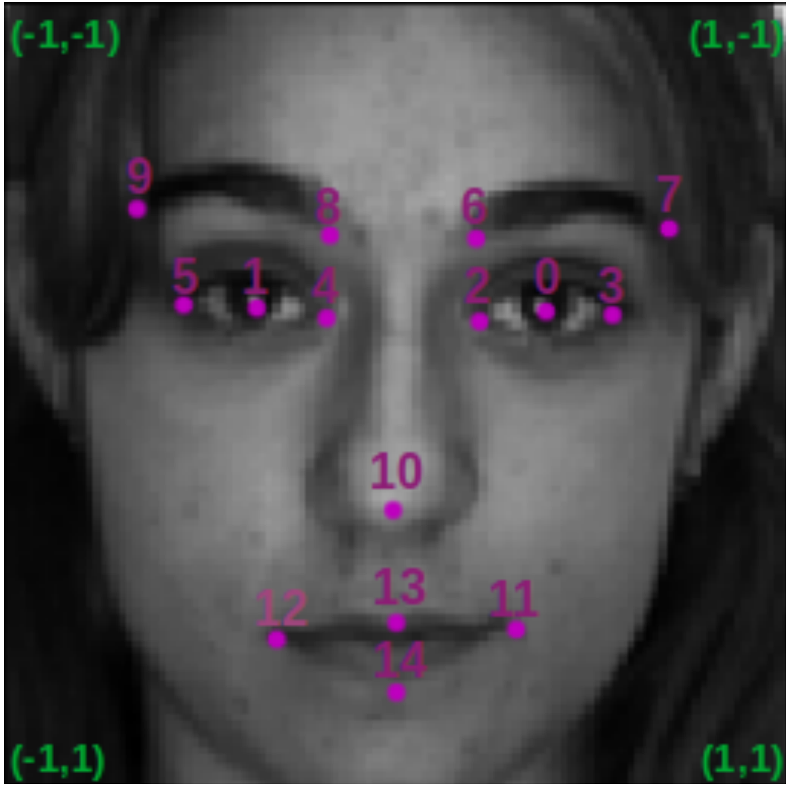
standard dropout: randomly drop units from the feature channels

dropped unit

Dropout for convolutional layers

- Fully convolutional networks exhibit strong spatial correlation
- Standard neuron-level dropout (i.e. randomly dropping a unit with a certain probability) does not improve performance in convolutional neural networks [1]
- Variant: SpatialDropout
  - randomly set entire feature maps to zero [2]

Submission: Facial Keypoints



Input:
(1, 96, 96) grayscale image

Output:
(2, 15) keypoint coordinates

Submission: Metric

```python
def evaluate_model(model, dataset):
    model.eval()
    criterion = torch.nn.MSELoss()
    dataloader = DataLoader(dataset, batch_size=1, shuffle=False)
    loss = 0
    for batch in dataloader:
        image, keypoints = batch["image"], batch["keypoints"]
        predicted_keypoints = model(image).view(-1,15,2)
        loss += criterion(
            torch.squeeze(keypoints),
            torch.squeeze(predicted_keypoints)
        ).item()
    return 1.0 / (2 * (loss/len(dataloader)))

print("Score:", evaluate_model(dummy_model, val_dataset))
```

Submission: Details

- Submission **Start: June 27**, 2020 12.00
- Submission **Deadline : July 03**, 2020 23.59
- Your model's **evaluation score** is all that counts!
  - Evaluation score: 1 / (2 * MSE)
  - A **score of at least 100** to pass the submission

Good luck &
see you next week
☺