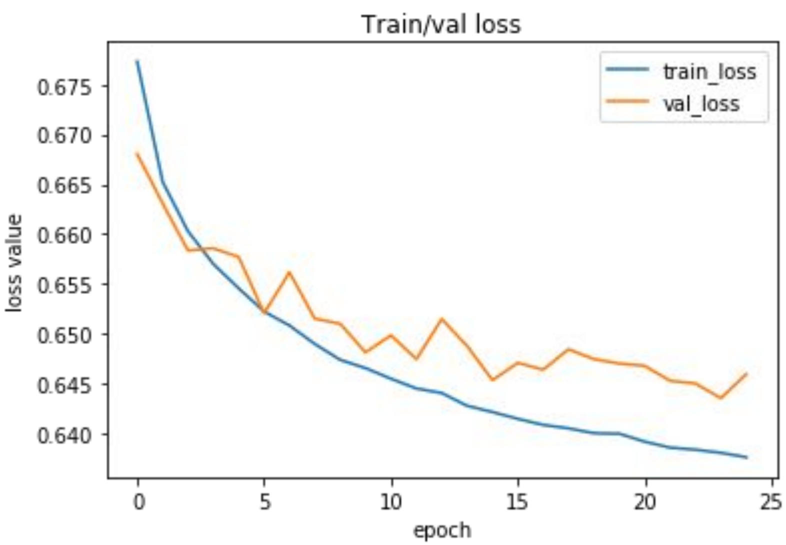
ATML Tutorial 05
Regularization

Advanced Topics in Machine Learning
17.03.2020
Adam Bielski

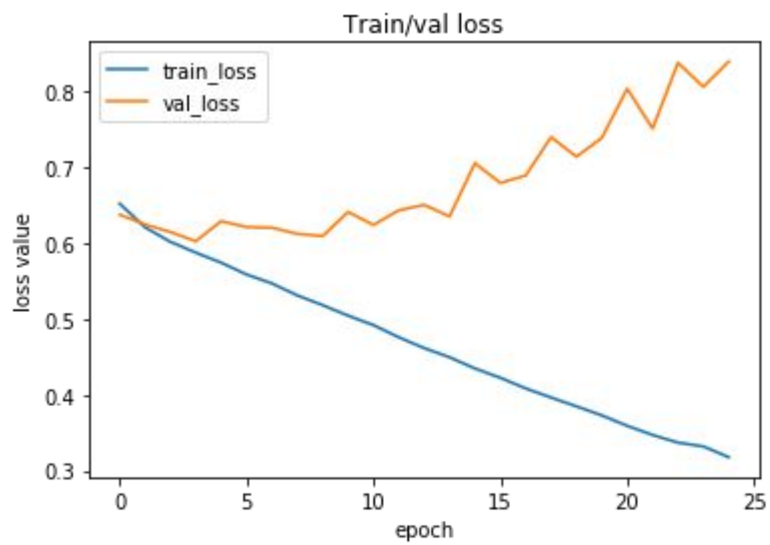
Overfitting

Tutorial 3: dogs vs cats classification

Linear model



MLP with 2 hidden layers



Underfitting: model is too simple to get good results

Overfitting: Model is too complex for our dataset

- **memorizes training examples**
- **doesn't generalize to new data**

Overfitting

- We want to use complex models that can learn better representations
- We can use regularization to reduce overfitting

Regularization

Option A: get more data => expensive, difficult

generate more data => **data augmentation**

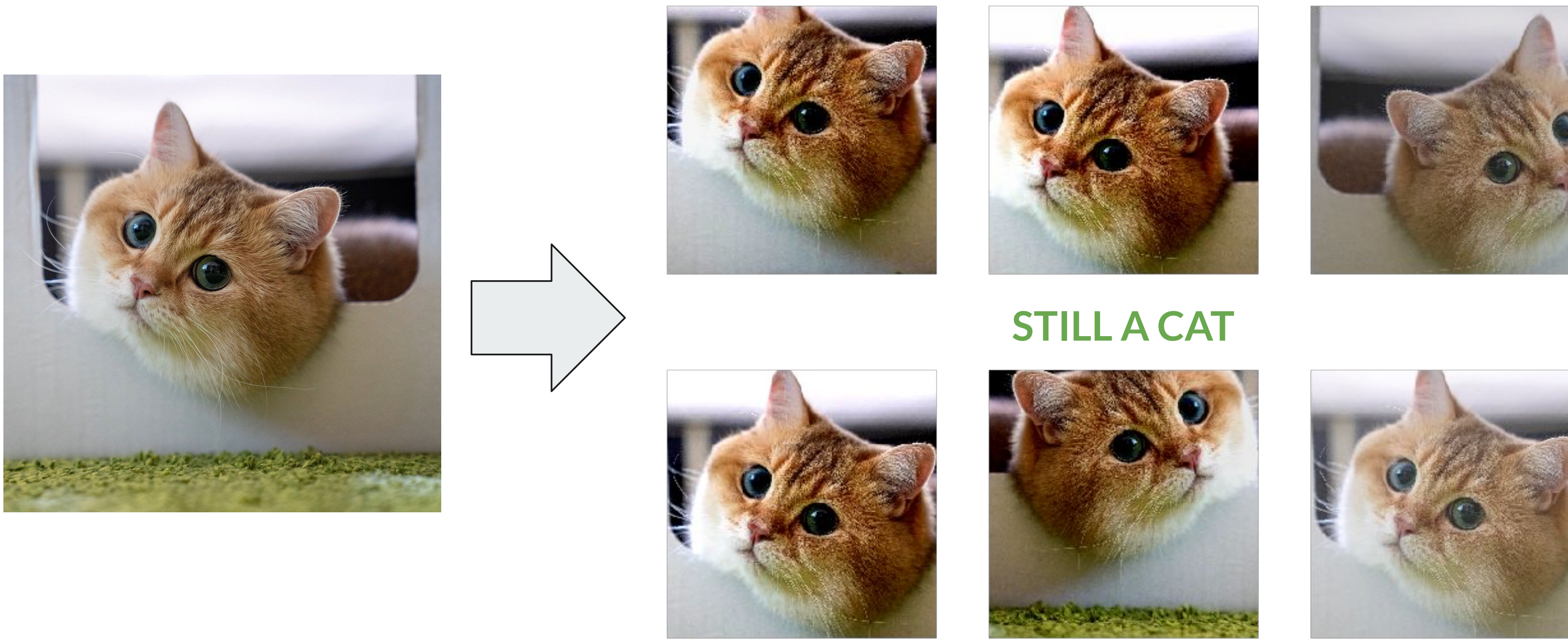
Option B: constrain your model

- make it more difficult to memorize the data

=> **L2 regularization, Dropout, Early stopping**

Data augmentation

Apply transformations to the data that preserves a label,
e.g. **Crop, Rotation, Affine Transformation, Color Jitter...**

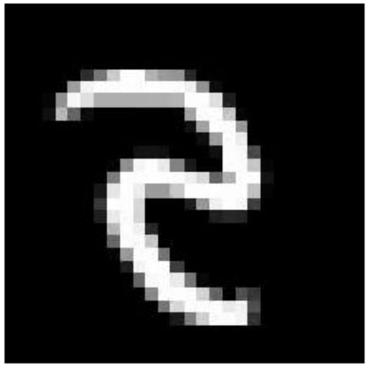


Data augmentation

Choose carefully, data-domain specific!

Horizontal Flip

STILL A CAT
- Label preserved



NOT A FIVE ANYMORE
- Label changed

Data augmentation - PyTorch

Many of them implemented in **torchvision** package
https://pytorch.org/docs/stable/torchvision/transforms.html

See: augmentation_transforms.ipynb

Google Colab:
https://colab.research.google.com/drive/1VUMkwTyubaaFC4lyL6FzOikJRSGu36Tt

https://tinyurl.com/uy3ptv7

Data augmentation - PyTorch

Example with ImageFolder dataset class

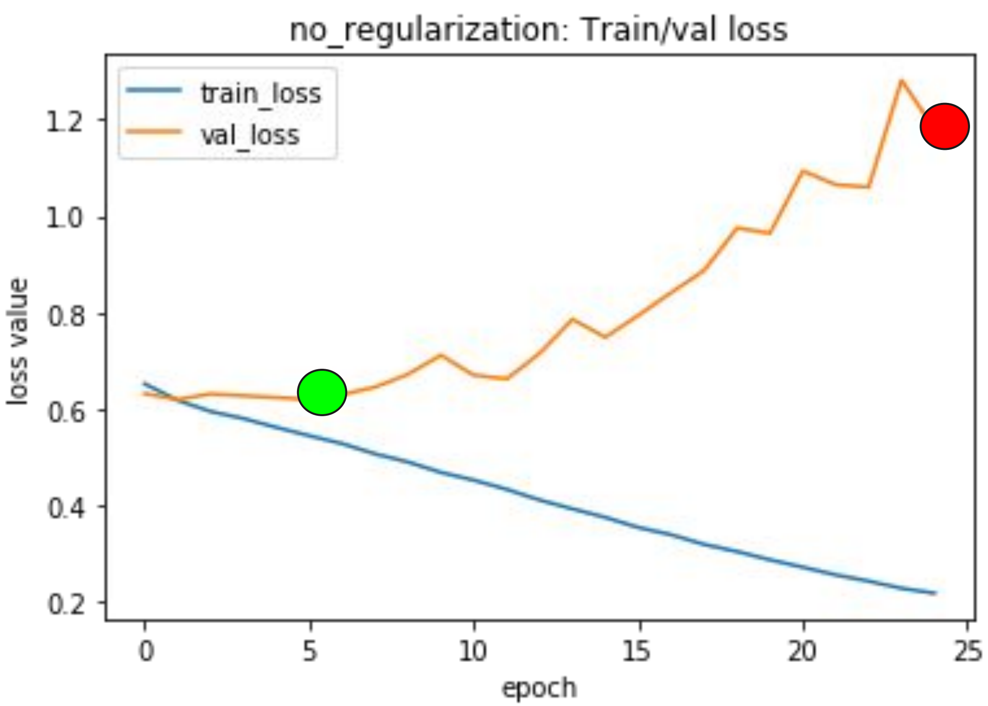
CLASS `torchvision.datasets.ImageFolder(root, transform=None, target_transform=None, loader=<function default_loader>)`

```python
from torchvision.datasets import ImageFolder
from torchvision.transforms import Resize, ToTensor, Normalize, Compose

target_size = (32, 32)
transforms = Compose([Resize(target_size),# Resizes image
                      ToTensor(),           # Converts to Tensor
                      Normalize(mean=(0.5, 0.5, 0.5,),
                                std=(0.5, 0.5, 0.5)), # scaling
                      ])

train_dataset = ImageFolder(data_dir, transform=transforms)
```

```python
from torchvision.datasets import ImageFolder
from torchvision.transforms import Resize, ToTensor, Normalize, Compose


transforms = Compose([Resize((40, 40)), # Resizes image
                      RandomCrop((32, 32)), # Crop 32x32 area
                      RandomHorizontalFlip(),
                    ToTensor(),           # Converts to Tensor

                    Normalize(mean=(0.5, 0.5, 0.5,),
                              std=(0.5, 0.5, 0.5)),
                    ])

train_dataset = ImageFolder(data_dir, transform=transforms)
```

Early stopping

Monitor validation loss / accuracy, save the model with the best value



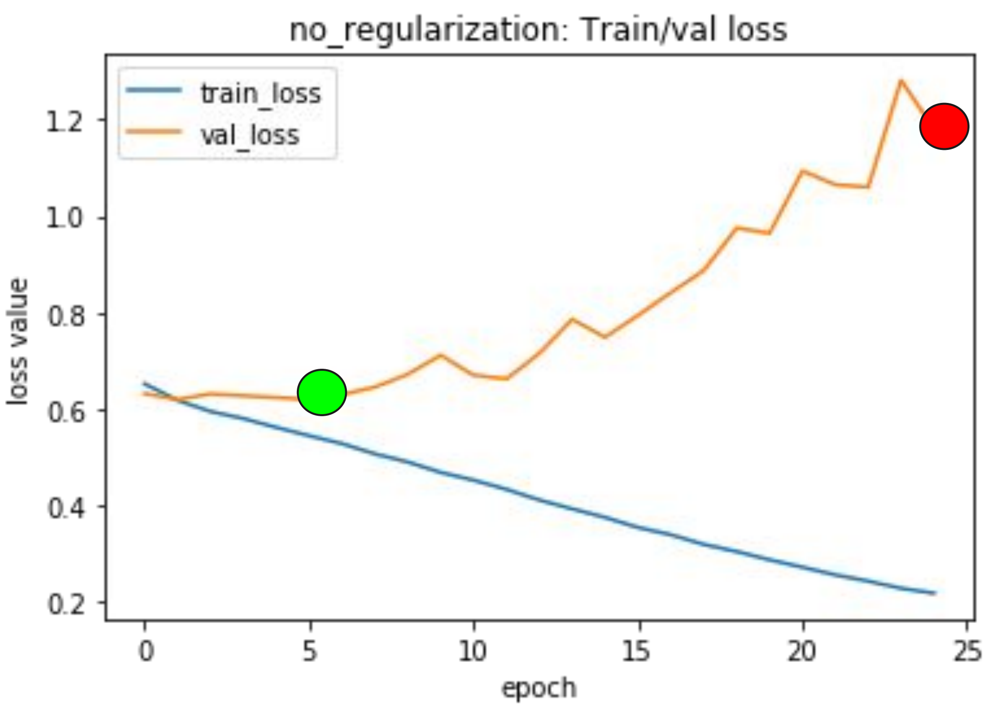
is better than ;

Monitor loss and get the model from

Stop the training if the validation loss starts increasing

Early stopping

Monitor validation loss / accuracy, save the model with the best value




```
best_val_loss = np.inf
best_model = None
patience = 5 # if no improvement after 5 epochs, stop tra
counter = 0


for epoch in range(n_epochs):
    ### Train for an epoch and evaluate on validation set
    ### (...)

    if val_loss < best_val_loss:
        best_val_loss = val_loss
        best_model = deepcopy(model)
        counter = 0
    else:
        counter += 1
    if counter == patience:
        print('No improvement for {} epochs;
              training stopped.'.format(patience))
        break
```

L2 Regularization / weight decay

Constrain magnitude of **weights** / **parameters** of the neural network by adding a term to the loss function

$$L = L_{cross-entropy} + \boxed{\frac{1}{2}\alpha \sum_i \theta_i^2}$$

L2 regularization

Theta - neural network weights
Alpha - regularization strength (if too big, all weights will shrink to 0 - network will not learn)

In PyTorch we can define it in the optimizer:

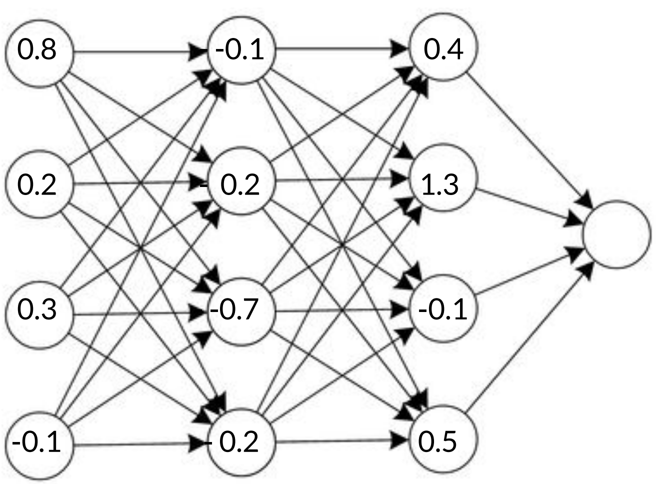
CLASS `torch.optim.SGD(params, lr=<required parameter>, momentum=0, dampening=0, weight_decay=0, nesterov=False)` [SOURCE]

^ We provide *alpha* here

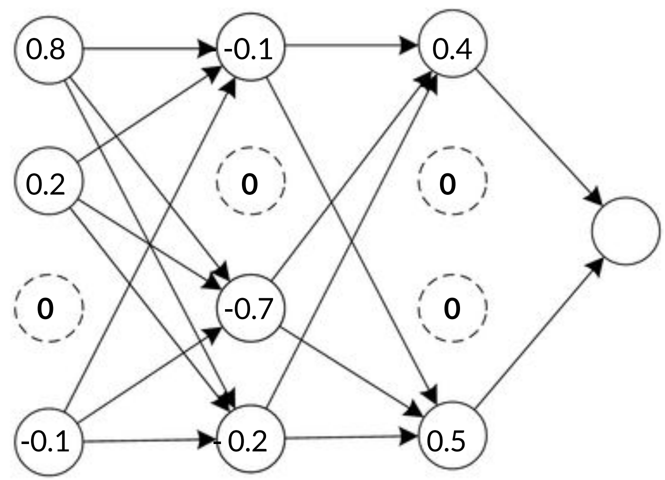
```
alpha = 0.001
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate, weight_decay=alpha)
```

Dropout

- Set hidden layer outputs to zero **in each iteration randomly; for training**
- Use all the outputs (do nothing) **for evaluation**



(a) Standard Neural Network



(b) Network after Dropout

Dropout

In PyTorch - **we add nn.Dropout() layers in model definition**

// probability of setting a unit to 0

```
class MLPModelDropout(nn.Module):

    def __init__(self, input_dim, hidden_dim, dropout_p=0.5):
        super(MLPModelDropout, self).__init__()
        self.layers = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Dropout(dropout_p),  ### Adding dropout layer
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Dropout(dropout_p),  ### Adding dropout layer
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Dropout(dropout_p),  ### Adding dropout layer
            nn.Linear(hidden_dim, 2))

    def forward(self, input):
        input = input.view(input.size(0), -1)
        return self.layers(input)
```

Dropout

```
model = MLPModelDropout(32*32*3, 128, 0.5)
model.train() # Sets all the layers to training mode - dropout is active
model.eval()  # Seta all the layers to evaluation mode - dropout is inactive (does nothing)
```

See dropout_layer.ipynb