Exercise 8: Autoencoder

Data Augmentation at Beginning

- Importance:
  - Data augmentation is a solution towards limited training data
  - Also improve generalization ability of your model.

- Two types of data augmentation:
  - Offline Augmentation
  - Online Augmentation

Data Augmentation

- Offline Augmentation:
  - As a pre-processing step to increase the size of the dataset. This is usually done when we have a small training dataset. In this case, the size of the augmented dataset is fixed.

- Online Augmentation:
  - Apply transformations in mini-batches and then feed it to the model. So the size of the augmented dataset that the model actually sees can be infinitely large.

Encoder

```python
class Encoder(nn.Module):

    def __init__(self, hparams, input_size=28 * 28, latent_dim=20):
        super().__init__()

        # set hyperparams
        self.latent_dim = latent_dim
        self.input_size = input_size
        self.hparams = hparams
        self.encoder = None

        ########################################################################
        # TODO: Initialize your encoder!                                       #
        ########################################################################

        self.encoder = nn.Sequential(
            nn.Linear(input_size, 500),
            nn.BatchNorm1d(500),
            nn.ReLU(),
            nn.Dropout(p=0.5),
            nn.Linear(500, 100),
            nn.BatchNorm1d(100),
            nn.ReLU(),
            nn.Dropout(p=0.5),
            nn.Linear(100, latent_dim),
            nn.BatchNorm1d(latent_dim),
            nn.ReLU()
        )

        ########################################################################
        #                           END OF YOUR CODE                           #
        ########################################################################
```

- Remark: This is a typical set up for fully-connected layers. You can also be creative here and come up with your own architecture ☺

Classifier

```python
class Classifier(pl.LightningModule):

    def __init__(self, hparams, encoder, train_set=None, val_set=None, test_set=None):
        super().__init__()
        # set hyperparams
        self.hparams = hparams
        self.encoder = encoder
        self.model = nn.Identity()
        self.data = {'train': train_set,
                     'val': val_set,
                     'test': test_set}
        ########################################################################
        # TODO: Initialize your classifier!                                    #
        # Remember that it must have the same inputsize as the outputsize      #
        # of your encoder                                                      #
        ########################################################################

        self.model = nn.Linear(20, 10)

        ########################################################################
        #                           END OF YOUR CODE                           #
        ########################################################################
```

- Remark: Here we show a very simple classifier, but the important thing to note here is that you have to match the input shape of the classifier to the output shape of your encoder implemented above.

Simple Encoder-Classifier Model

```
########################################################################
# TODO: Define your hyper parameters here!                             #
########################################################################
hparams = {
    "batch_size": 128,
    "learning_rate": 1e-2
}
########################################################################
#                           END OF YOUR CODE                           #
########################################################################
```

- Remark: With the given hyperparameters, our Encoder-Classifier model can reach an accuracy around 70%

Autoencoder

- Model Architecture:
  - As suggested in the exercise notebook, the simplest way is to have a symmetric architecture which ensure that the latent information can be reconstructed properly.

- Reconstruction Loss:
  - In this exercise, we use the mean squared error loss between our input pixels and the output pixels. Please think what would be the potential drawbacks of this type of loss. ☺

Decoder

```python
class Decoder(nn.Module):

    def __init__(self, hparams, latent_dim=20, output_size=28 * 28):
        super().__init__()

        # set hyperparams
        self.hparams = hparams
        self.decoder = None

        ########################################################################
        # TODO: Initialize your decoder!                                       #
        ########################################################################

        self.decoder = nn.Sequential(
            nn.Linear(latent_dim, 100),
            nn.BatchNorm1d(100),
            nn.Dropout(p=0.5),
            nn.ReLU(),
            nn.Linear(100, 500),
            nn.BatchNorm1d(500),
            nn.Dropout(p=0.5),
            nn.ReLU(),
            nn.Linear(500, output_size)
        )

        ########################################################################
        #                           END OF YOUR CODE                           #
        ########################################################################
```

- Remark: As suggested before, we will mirror the architecture of the encoder to construct the decoder.

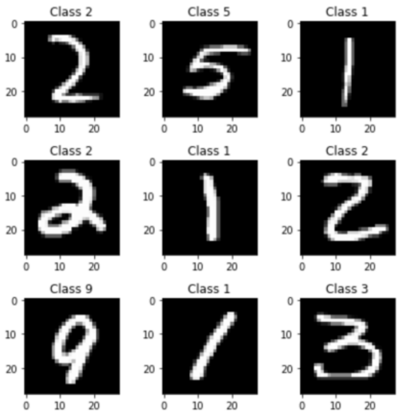
Autoencoder Training

```
########################################################################
# TODO: Define your hyperparameters here!                              #
########################################################################
hparams = {
    "batch_size": 128,
    "learning_rate": 5e-3
}
########################################################################
#                           END OF YOUR CODE                           #
########################################################################
```

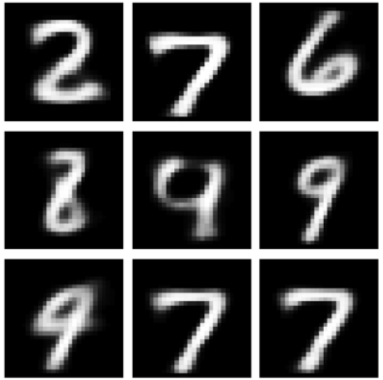
```
########################################################################
# TODO: Define your trainer! Don't forget the logger.                  #
########################################################################

ae_trainer = pl.Trainer(
    max_epochs=30,
    gpus=1 if torch.cuda.is_available() else None,
    logger=ae_logger
)

########################################################################
#                           END OF YOUR CODE                           #
########################################################################
```

- Remark: The hyperparameter and the trainer here is similar to our previous training of the encoder-classifier model.

Reconstruction Analysis



Original Digits

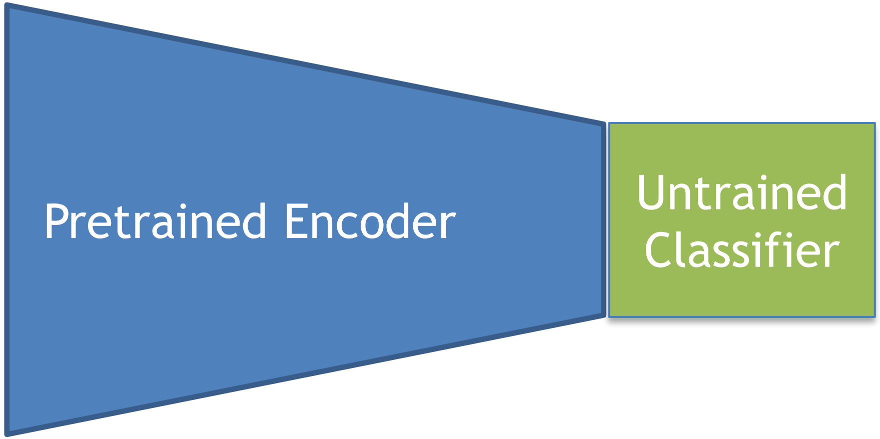


Reconstructed Digits

- We can see that the reconstructed digits look similar to the original ones, but they are more blurry.
- The reason of this are mainly two aspects:
  - First, out latent dimension might be too small so that we lost some useful information
  - Second, the L2 reconstruction loss that we use essentially converge to a mean value, which we would lose the sharpness.

Transfer Learning

- Now, we come to the most important part of this exercise, which we take the pretrained encoder and our classifier to build our final model, and trained on only the labelled data



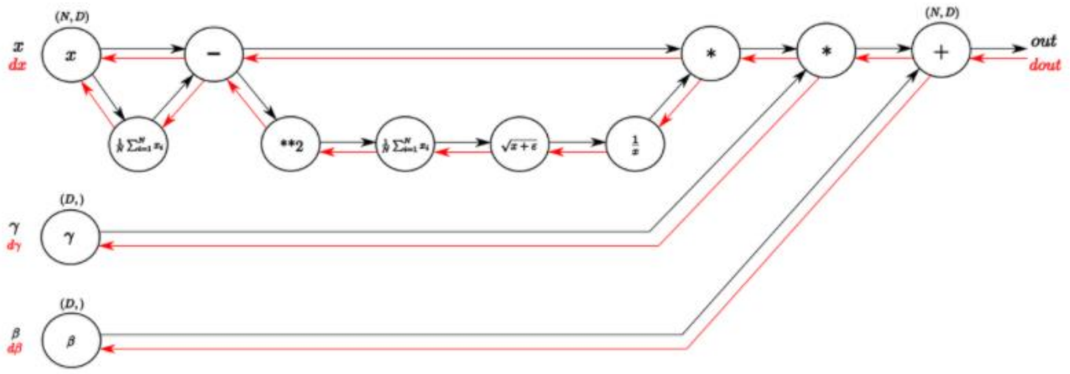
Transfer Learning

```
########################################################################
# TODO: Define your hyper parameters here!                             #
########################################################################
hparams = {
    "batch_size": 256,
    "learning_rate": 1e-2
}
########################################################################
#                           END OF YOUR CODE                           #
########################################################################
```

```
########################################################################
# TODO: Define your trainer! Don't forget the logger.                  #
########################################################################

trainer = pl.Trainer(
    max_epochs=50,
    gpus=1 if torch.cuda.is_available() else None
)

########################################################################
#                           END OF YOUR CODE                           #
########################################################################
```

- Remarks: With the example hyperparameters, we can reach an accuracy at around 80%

Batch Normalization (Optional)



Source: https://kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalization-layer.html

- Remarks: This is a computational graph of the forward pass and the backward pass of the batch normalization. It could help you better understand the flow of computation

BatchNorm-forward

```python
if mode == 'train':
    #######################################################################
    # TODO: Look at the training-time forward pass implementation for batch#
    # normalization.                                                      #
    #######################################################################
    sample_mean = np.mean(x, axis=0)
    x_minus_mean = x - sample_mean
    sq = x_minus_mean ** 2
    var = 1. / N * np.sum(sq, axis=0)
    sqrtvar = np.sqrt(var + eps)
    ivar = 1. / sqrtvar
    x_norm = x_minus_mean * ivar
    gammax = gamma * x_norm
    out = gammax + beta
    running_var = momentum * running_var + (1 - momentum) * var
    running_mean = momentum * running_mean + (1 - momentum) * sample_mean
    cache = (out, x_norm, beta, gamma, x_minus_mean, ivar, sqrtvar, var, eps

    #######################################################################
    #                           END OF YOUR CODE                          #
    #######################################################################
elif mode == 'test':
    #######################################################################
    # TODO: Look at the test-time forward pass for batch normalization.   #
    #######################################################################
    x = (x - running_mean) / np.sqrt(running_var)
    out = x * gamma + beta
    #######################################################################
    #                           END OF YOUR CODE                          #
    #######################################################################
```

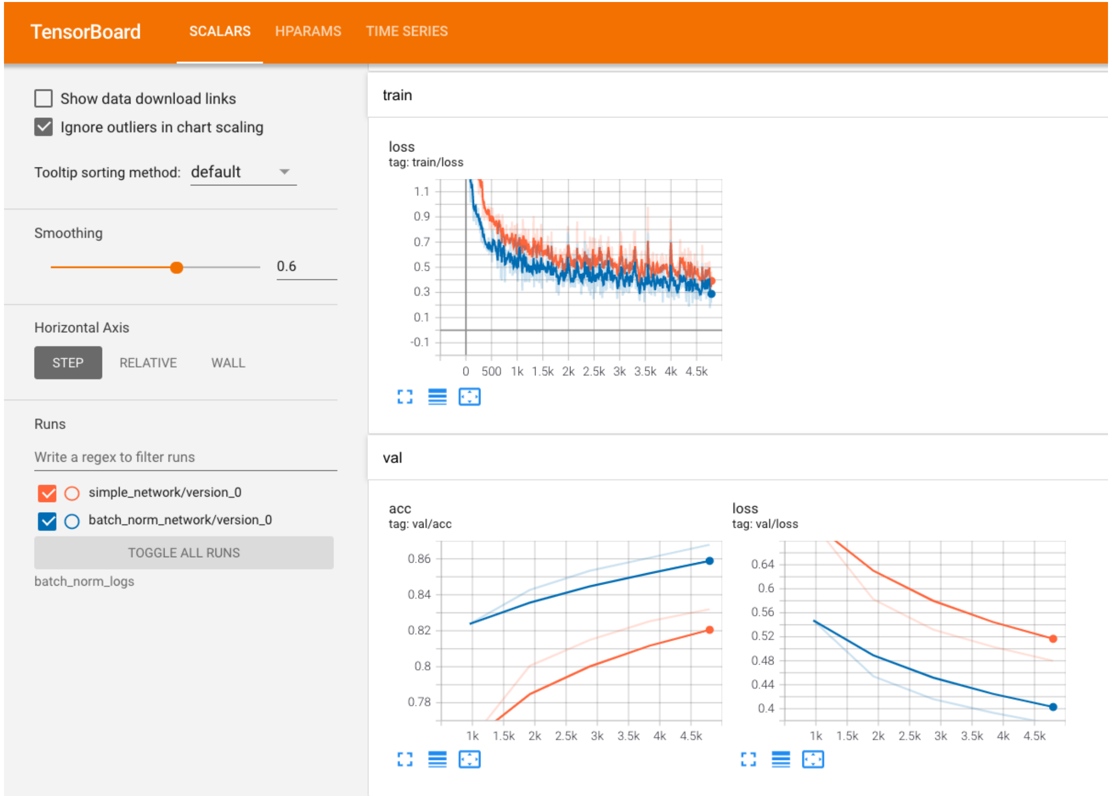
- Remarks: Note the difference between training phase and testing phase

BatchNorm-backword

```
##########################################################################
# TODO: Implement the backward pass for batch normalization.            #
##########################################################################
N, D = dout.shape
out, x_norm, beta, gamma, xmu, ivar, sqrtvar, var, eps = cache
dxnorm = dout * gamma
divar = np.sum(dxnorm * xmu, axis=0)
dxmu1 = dxnorm * ivar
dsqrtvar = -1. / (sqrtvar ** 2) * divar
dvar = 0.5 * 1. / np.sqrt(var + eps) * dsqrtvar
dsq = 1. / N * np.ones((N, D)) * dvar
dxmu2 = 2 * xmu * dsq
dx1 = dxmu1 + dxmu2
dmean = -1. * np.sum(dx1, axis=0)
dx2 = 1. / N * np.ones((N, D)) * dmean
dx = dx1 + dx2
dbeta = np.sum(dout, axis=0)
dgamma = np.sum(dout * x_norm, axis=0)
##########################################################################
#                           END OF YOUR CODE                            #
##########################################################################
```

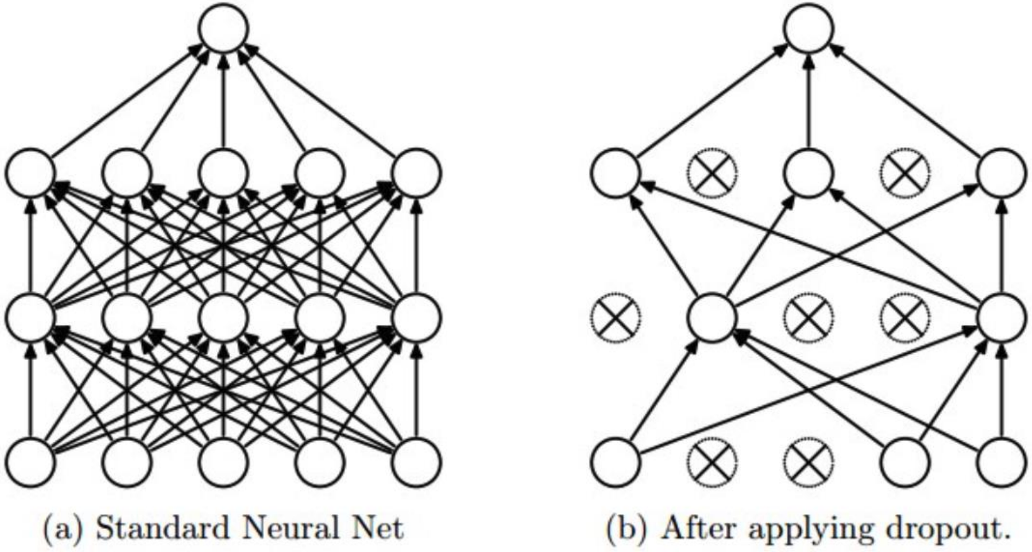
- Remarks: Utilize the computational graph of batch normalization will help you understand the backward pass ☺

BatchNorm-Training Results



- Remarks: As can be seen from the tensorboard, the model with batch normalization (blue curve) results in better performance on both training and validation set

Dropout (Optional)



- Remarks: Dropout is a regularization technique for neural networks by randomly setting some features to zero during the forward pass

Dropout-forward

```python
if mode == 'train':
    #######################################################################
    # TODO: Implement the training phase forward pass for inverted dropout. #
    # Store the dropout mask in the mask variable.                          #
    #######################################################################
    mask = (np.random.rand(*x.shape) > p) / (1 - p)
    out = x * mask
    #######################################################################
    #                           END OF YOUR CODE                          #
    #######################################################################
elif mode == 'test':
    #######################################################################
    # TODO: Implement the test phase forward pass for inverted dropout.   #
    #######################################################################
    out = x
    #######################################################################
    #                           END OF YOUR CODE                          #
    #######################################################################
```

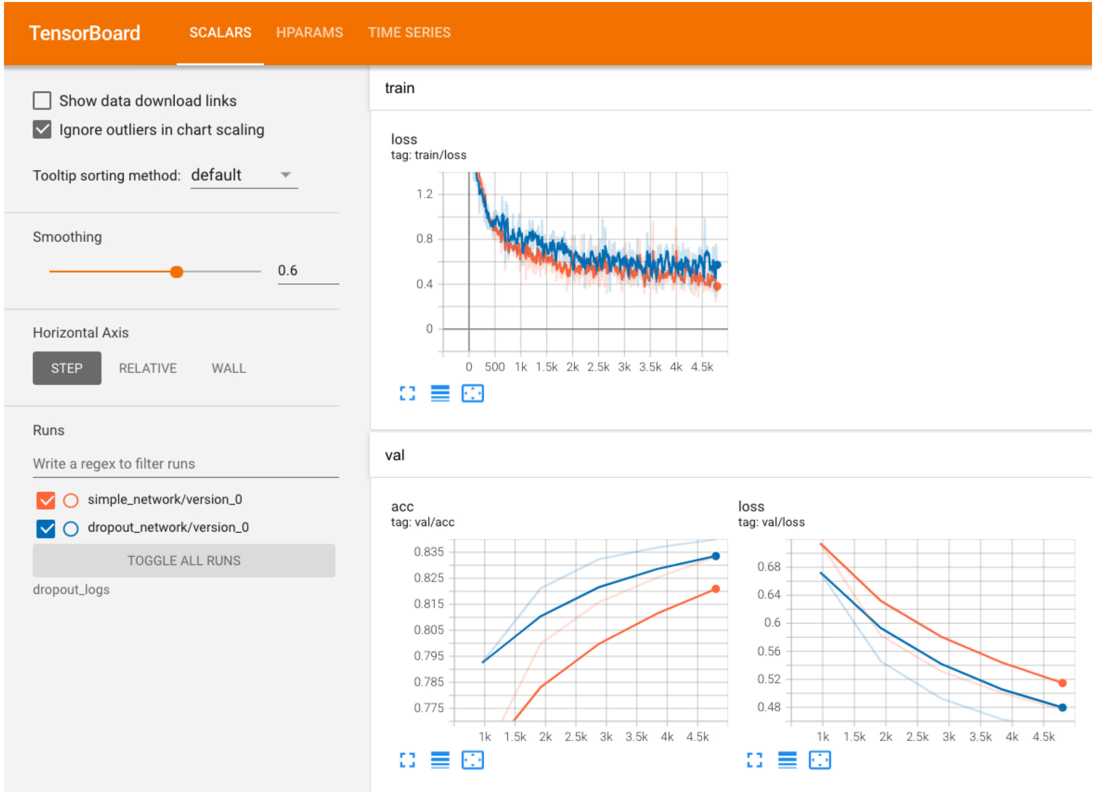
- Remarks: Note that we will not ‘drop’ neurons at test time

Dropout-backward

```
if mode == 'train':
    #######################################################################
    # TODO: Implement the training phase backward pass for inverted dropout.
    #######################################################################
    dx = dout * mask
    #######################################################################
    #                          END OF YOUR CODE                           #
    #######################################################################
elif mode == 'test':
    dx = dout
```

- Remarks: Note the difference between training phase and testing phase that we don't apply dropout at test time

Dropout-Training Results



- Remarks: As can be seen from the tensorboard, the model with dropout has slightly higher training loss, but the model would perform better on the validation set.

Questions? Piazza

☺