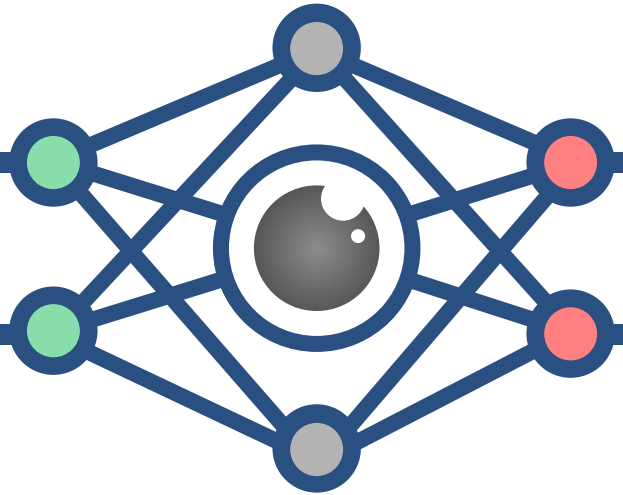
CS3485

Deep Learning for Computer Vision



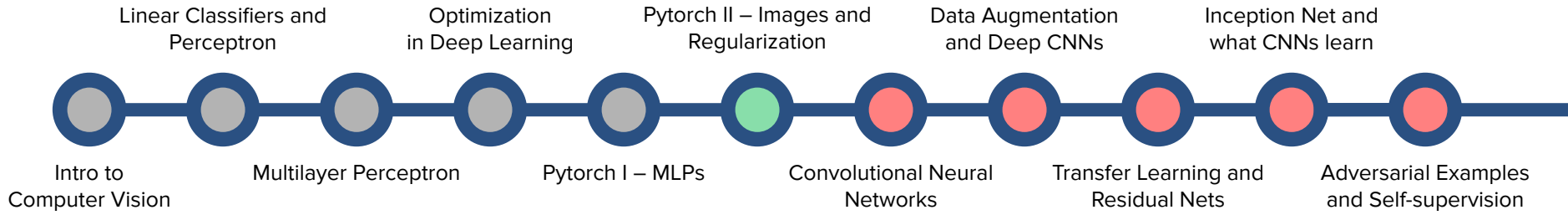
Lec 6: Pytorch II – Images and Regularization

Announcements

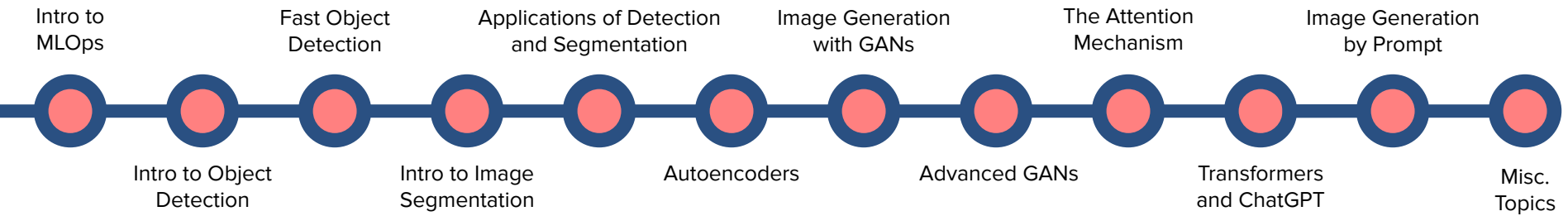
- Bowdoin's HPC:
  - You can run Large Language Models on the HPC! Here's the tutorial for doing that for different language models.
- No lab today. The next one will be released next Tuesday.

(Tentative) Lecture Roadmap

Basics of Deep Learning

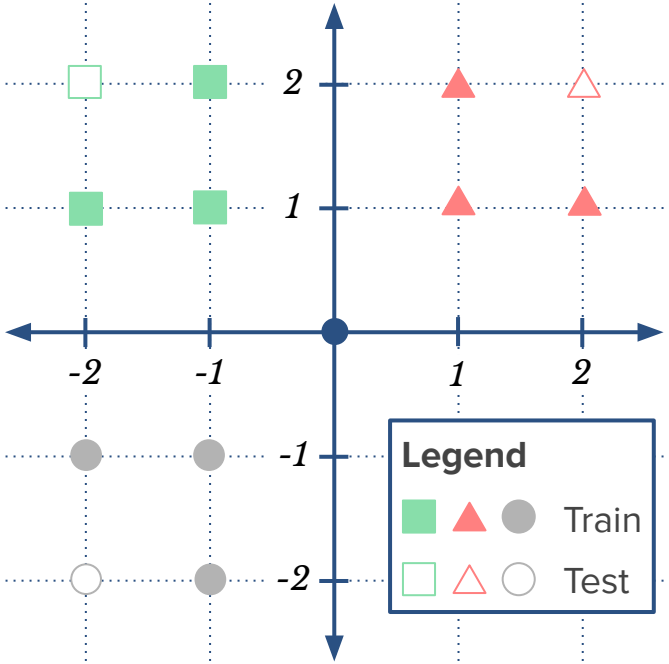


Deep Learning and Computer Vision in Practice



MLPs in Pytorch

- Last time we saw how to write a simple Pytorch scripts and use it in a classification problem (*on the right*).
- However, despite using SGD, we were not mini-batches and, truth be told the data was quite simple to work with.
- Today we'll improve our pytorch implementation from last time and cover the important topic of **regularization** in Deep Learning.

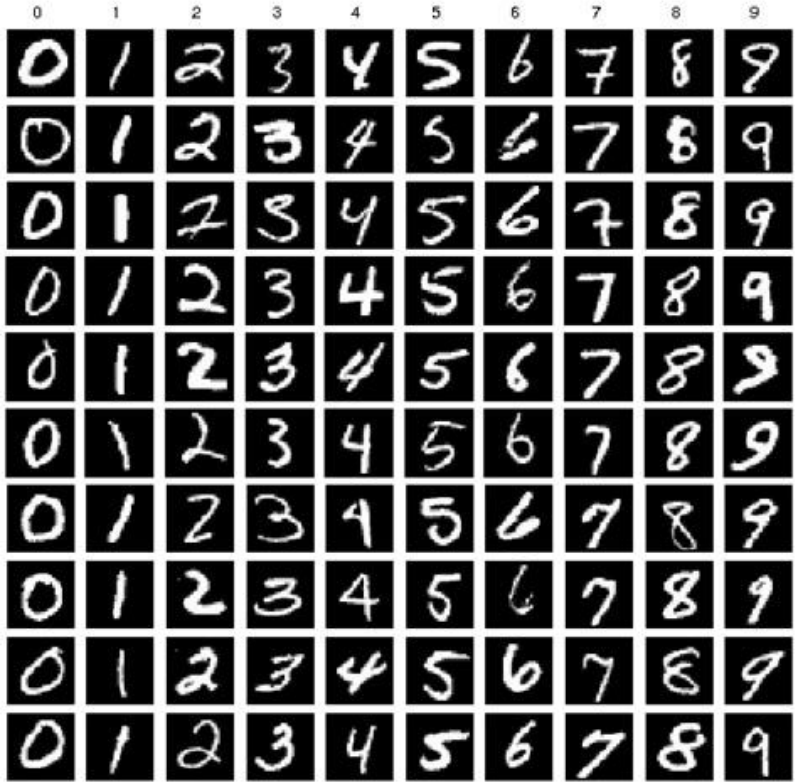


Our dataset from last time

PyTorch

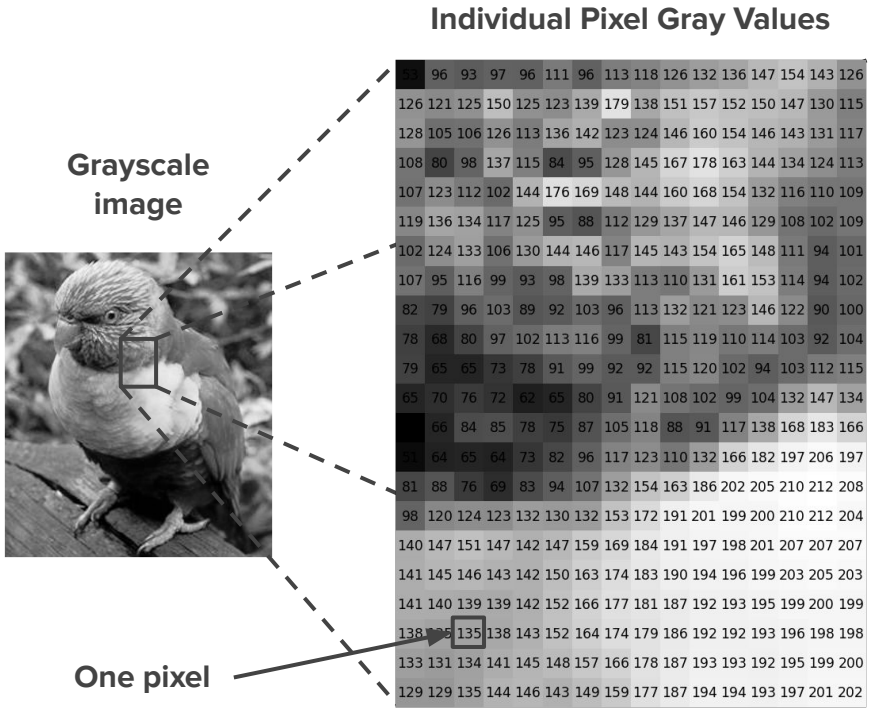
Making more complex networks

- The toy dataset we last time before isn't a good example of realistic data.
- A more realistic (and more convenient to Computer Vision) dataset in the **MNIST handwritten digits.**
- The database contains 28×28 grayscale images representing the digits 0 through 9 (some depicted on the right).
- The data is split into two subsets, with 60000 images for training and 10000 images for testing.
- Before we go there, we should ask ourselves: **what is an image**?



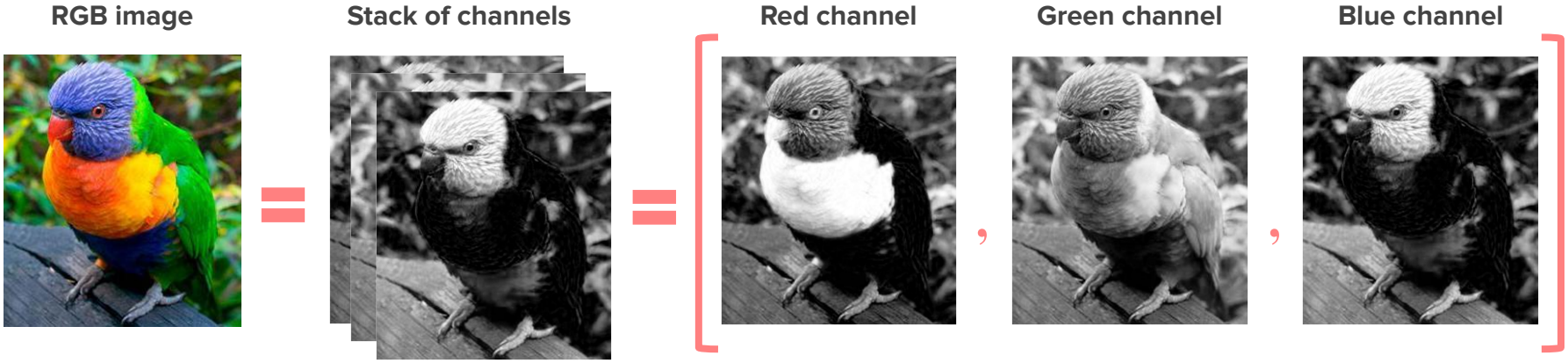
What is an image?

- An **image** as a data structure is simply a matrix (or matrices) of integer numbers ranging from 0 to 255 (one byte of data).
- **Pixels** are the individual subdivisions of an image, each with one sole color.
- A **grayscale image** is an image that only needs one of these matrices to represent its data.
- In a grayscale image, each of its pixels is colored with a **shade of gray,** whose color ranges from **black** (value 0) to **white** (value 255).



What is an image?

- For colored images, we make each individual pixel contain 3 integers, each in $[0, 255]$.
- These values represent the the intensities of Red, Blue and Green (RGB)* in that pixel, in this order, giving the name for thai kind of image an **RGB image**.
- Each **color channel** (R, G or B) is then represented by a gray scale image and the full image is the stacking of its channels.



* There are other ways to define a colored image that is not RGB (ex.: CYMK = Cyan, Yellow, Magenta and Black).

Loading the data

- With that out of the way, we can now (down)load the MNIST dataset, which PyTorch provides (along with other datasets) in the `torchvision` library.

```
from torchvision import datasets

data_folder = '~/data/MNIST'
mnist_train = datasets.MNIST(data_folder, download=True, train=True)
X_train, y_train = mnist_train.data, mnist_train.targets
```

- We also create a class for our dataset that inherits from Pytorch's `Dataset`.
- Note that two pre-processing actions take place on the data:
  - **Rescaling**: it makes all data values be in the range [0, 1].
  - **Reshaping**: turn images into vectors.

```
from torch.utils.data import Dataset
class MNISTDataset(Dataset):
    def __init__(self, x, y):
        x = x.float()/255      # Data rescaling
        x = x.view(-1,28*28)   # Data reshaping
        self.x, self.y = x, y
    def __len__(self):
        return len(self.x)
    def __getitem__(self, ix):
        x, y = self.x[ix], self.y[ix]
        return x.to(device), y.to(device)
```

Visualizing the data

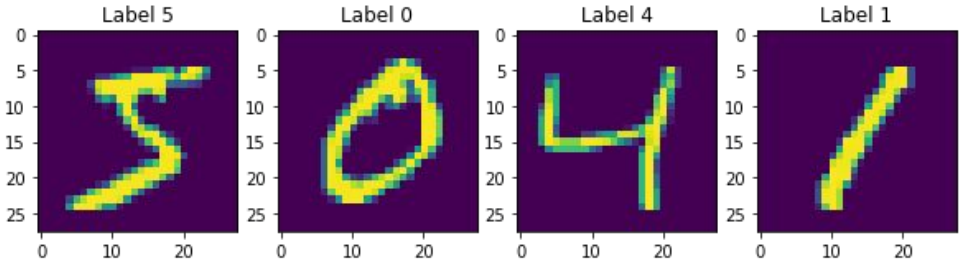
- It is always a good practice to check how the data looks like before working on it.

```
print(x_train.shape)
print(y_train.shape)
```

```
torch.Size([60000, 28, 28])
torch.Size([60000])
```

- This is an example of how PyTorch organizes the data dimensions: **(N,C,H,W)**
  - First the number of datapoints, (**N**); then the number of channels in each point, (**C**); then the height (**H**) and width of each point (**W**).
  - If either dimension (`N`, `C`, `H` or `W`) is just 1, it won't show up.
- We can check the data itself. Here are the first *4* data points and their true labels:

```
import matplotlib.pyplot as plt
plt.figure(figsize=(10,3))
for i in range(4):
   plt.subplot(1,4,i+1)
   plt.imshow(x_train[i])
   plt.title(f"Label {y_train[i]}")
plt.show()
```



Dataloader

- We then instantiate an `MNISTDataset` using the data we just downloaded:

```
train_dataset = MNISTDataset(x_train, y_train)
```

- Now we can use that dataset object in conjunction to what pytorch calls a **Dataloader**:

```
from torch.utils.data import DataLoader
train_dl = DataLoader(train_dataset, batch_size=32, shuffle=True)
```

- In our context, a `DataLoader` object acts like a Python generator that yields a set of datapoints and their corresponding labels (a mini-batch) at a time.
- The number of data points it yields is controlled by `batch_size`.
- The `shuffle` parameter makes the loader shuffle the data once all its batches were yielded, in order to produce new batches for the next time it has to yield data.
- This object will coordinate the usage of mini-batches in our network learning process.

Network, Optimizers and Loss

- As before, we define and instantiate a similar NN, now with *1000* hidden units:

```
class MNISTNeuralNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.input_to_hidden_layer = nn.Linear(28 * 28, 1000) # The input size is the number Of pixels
        self.hidden_layer_activation = nn.ReLU()              # in an individual image.
        self.hidden_to_output_layer = nn.Linear(1000, 10)
    def forward(self, x):
        x = self.input_to_hidden_layer(x)
        x = self.hidden_layer_activation(x)
        x = self.hidden_to_output_layer(x)
        return x
model = MNISTNeuralNet().to(device)
```

- We'll use Cross Entropy again and, now, the ADAM optimizer (with learning rate of *0.01*):

```
from torch.optim import Adam
opt = Adam(model.parameters(), lr=1e-2)
loss_func = nn.CrossEntropyLoss()
```

Training the network

- This time, we'll create a function that performs those four typical in Pytorch training:

```
def train_batch(x, y, model, opt, loss_fn):
    model.train() # We'll see the meaning of this line later today

    opt.zero_grad()  # Flush memory
    batch loss = loss func(model(x), y)  # Compute loss
    batch_loss.backward()  # Compute gradients
    opt.step()  # Make a GD step
    return batch_loss.detach().cpu().numpy() # Removes grad, sends data to cpu, converts tensor to array
```

- We'll also keep track of the accuracy of our model using the function:

```
@torch.no_grad() # This decorator is used to tell PyTorch that nothing here is used for training
def accuracy(x, y, model):
    model.eval() # We'll see the meaning of this line later today

    prediction = model(x) # Check model prediction
    argmaxes = prediction.argmax(dim=1) # Compute the predicted labels for the batch
    s = torch.sum((argmaxes == y).float())/len(y) # Compute accuracy
    return s.cpu().numpy()
```

Training the network

- Now let's train our network!

```
losses, accuracies, n_epochs = [], [], 5
for epoch in range(n_epochs):
   print(f"Running epoch {epoch + 1} of {n_epochs}")

   epoch_losses, epoch_accuracies = [], []
   for batch in train_dl:
       x, y = batch
       batch_loss = train_batch(x, y, model, opt, loss_func)
       epoch_losses.append(batch_loss)
   epoch_loss = np.mean(epoch_losses)

   for batch in train_dl:
       x, y = batch
       batch_acc = accuracy(x, y, model)
       epoch_accuracies.append(batch_acc)
   epoch_accuracy = np.mean(epoch_accuracies)

   losses.append(epoch_loss)
   accuracies.append(epoch_accuracy)
```

Training the network

- Now let's train our network!

```
losses, accuracies, n_epochs = [], [], 5
for epoch in range(n epochs):
   print(f"Running epoch {epoch + 1} of {n_epochs}")

   epoch losses, epoch accuracies = [], []
   for batch in train_dl:
       x, y = batch
       batch_loss = train_batch(x, y, model, opt, loss_func)
       epoch_losses.append(batch_loss)
   epoch_loss = np.mean(epoch_losses)

   for batch in train_dl:
       x, y = batch
       batch_acc = accuracy(x, y, model)
       epoch_accuracies.append(batch_acc)
   epoch_accuracy = np.mean(epoch_accuracies)

   losses.append(epoch_loss)
   accuracies.append(epoch_accuracy)
```

Runs gradient descent on each batch (of size 32) at a time.

Computes the classification accuracy on the training data also using the same Dataloader

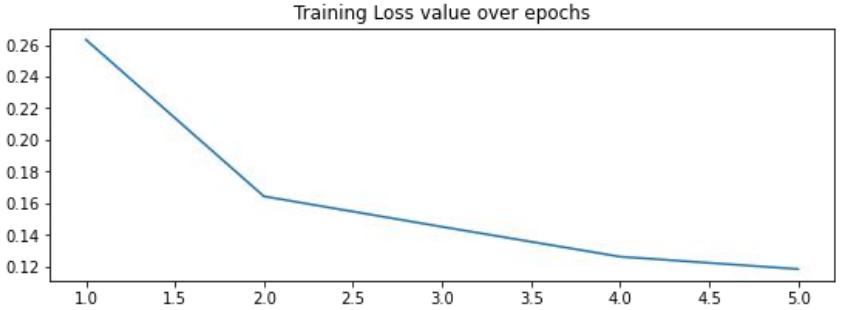
Keeps track of losses and training accuracy.

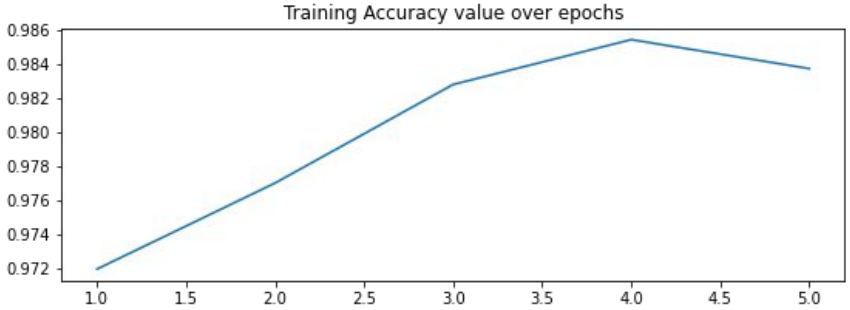
Plotting the training statistics

- We plot our train and test performance over epochs (*your results can be slightly different*):

```
import matplotlib.pyplot as plt

epochs = np.arange(n_epochs) + 1
plt.figure(figsize=(20,3))
plt.subplot(121)
plt.title('Training Loss value over epochs')
plt.plot(epochs, losses)
plt.subplot(122)
plt.title('Training Accuracy value over epochs')
plt.plot(epochs, accuracies)
```





Testing the model

- Finally we test the learned model on the test data. First we load the data and create a dataset object and a dataloader with it:

```
mnist_test = datasets.MNIST(data_folder, download=True, train=False)
x_test, y_test = mnist_test.data, mnist_test.targets

test_dataset = MNISTDataset(x_test, y_test)
test_dl = DataLoader(test_dataset, batch_size=32, shuffle=True)
```

  and then we compute the accuracy of the trained model on that dataset:

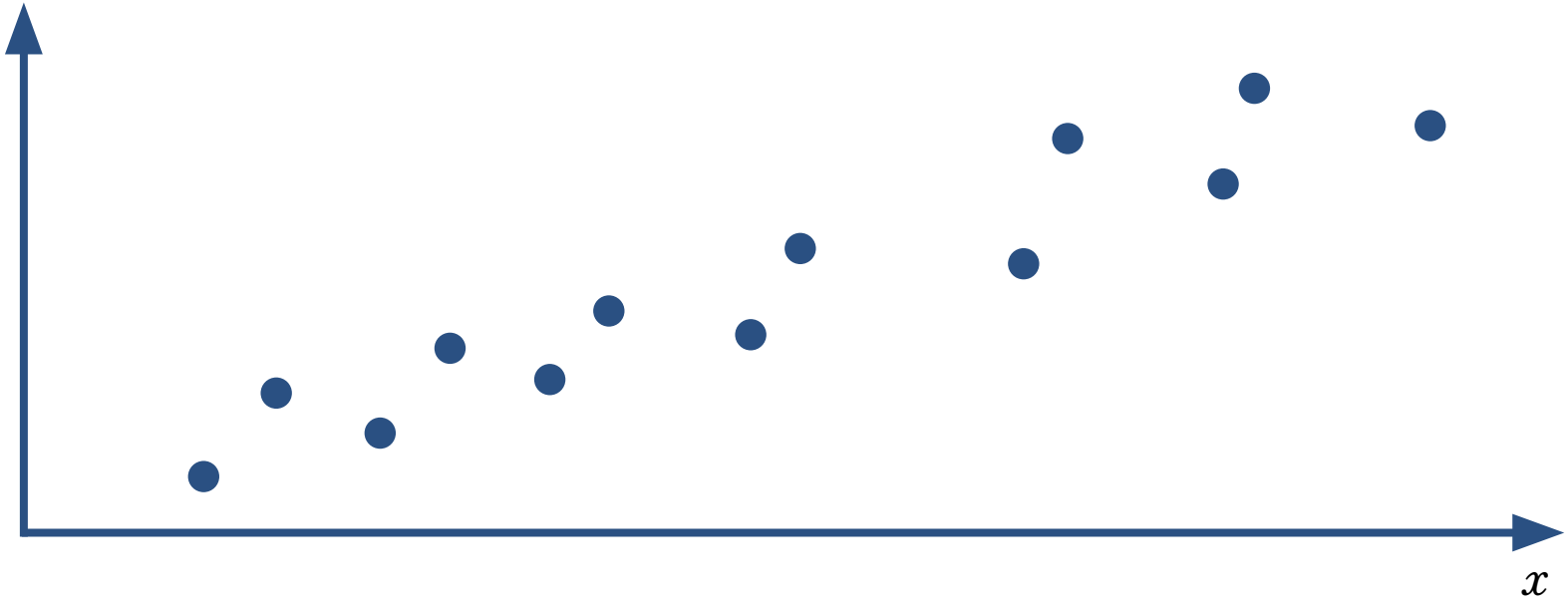
```
epoch_accuracies = []
for batch in test_dl:
   x, y = batch
   epoch_accuracies.append(accuracy(x, y, model))

print(f"Test accuracy: {np.mean(epoch_accuracies)}")
```

```
Test accuracy: 0.9637579917907715
```

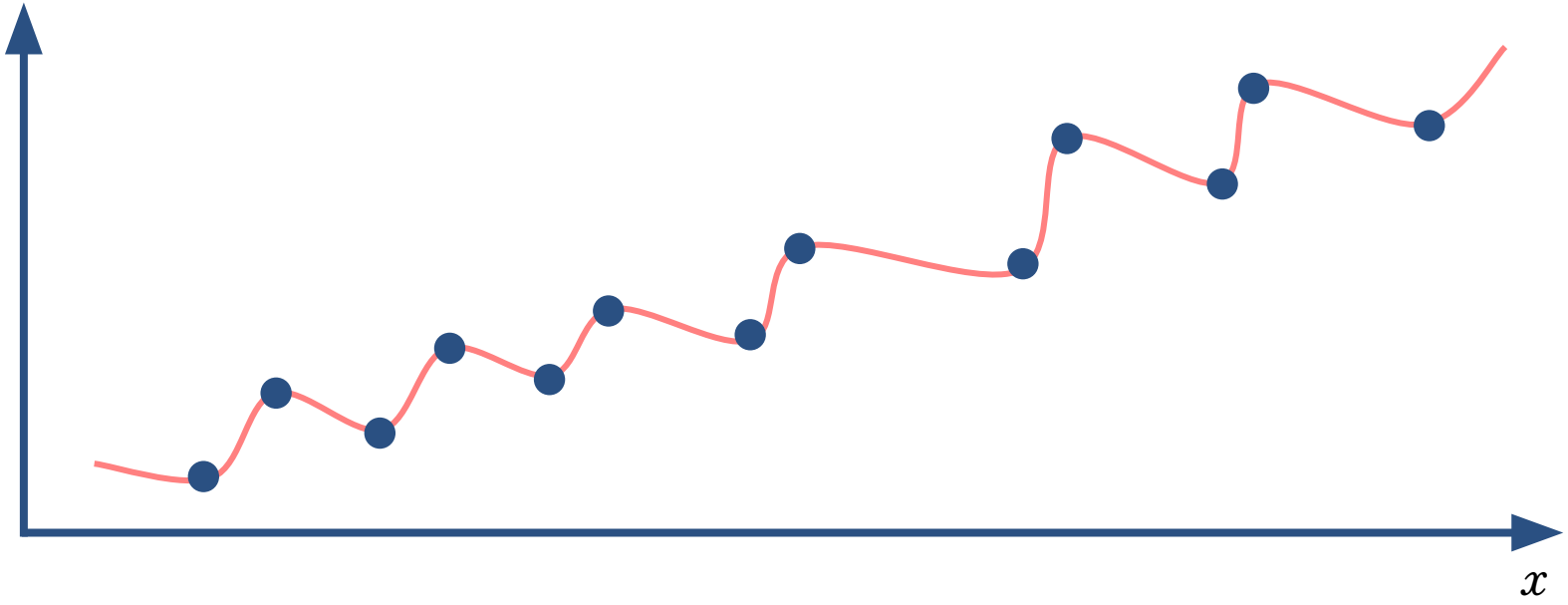
Regularization in Deep Learning

- Not bad: 96% of accuracy! But we can try to improve it with **Regularization**! But, first: if you were to draw the function that generated the points below, how would it look like?



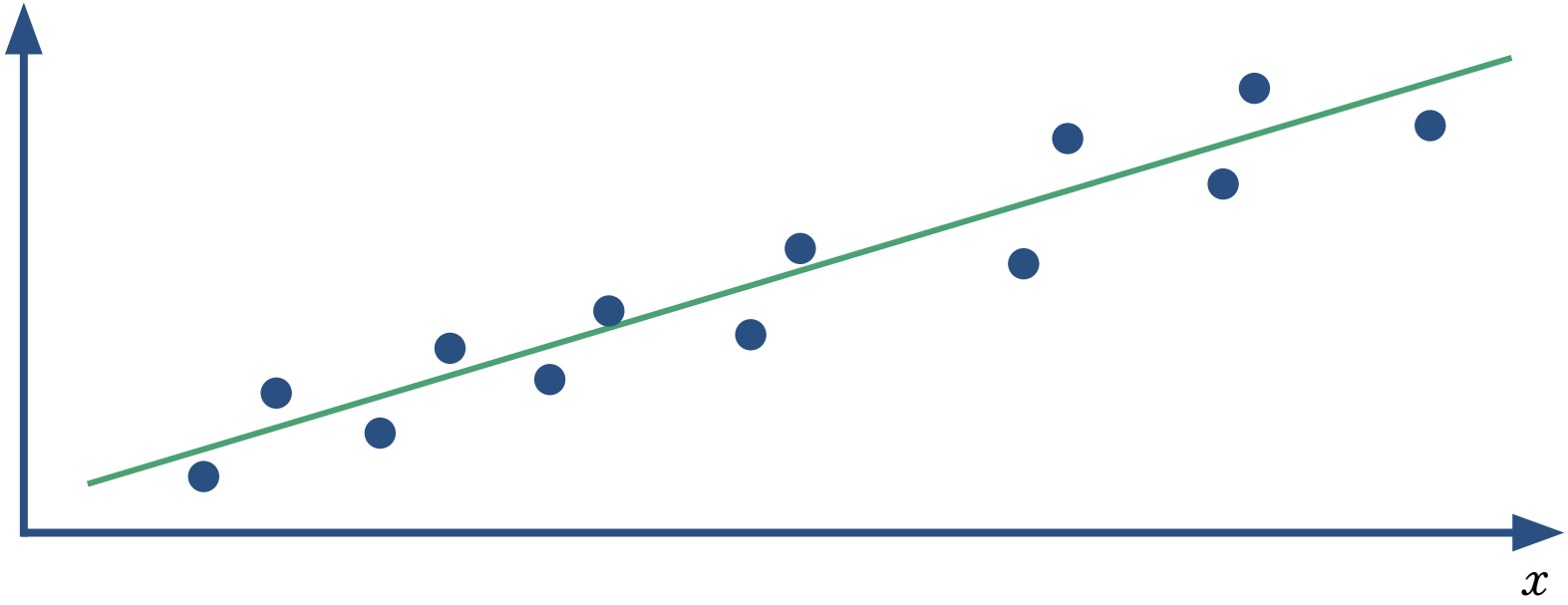
Regularization in Deep Learning

- A possible solution is the following, where we made sure that the function went through all points:



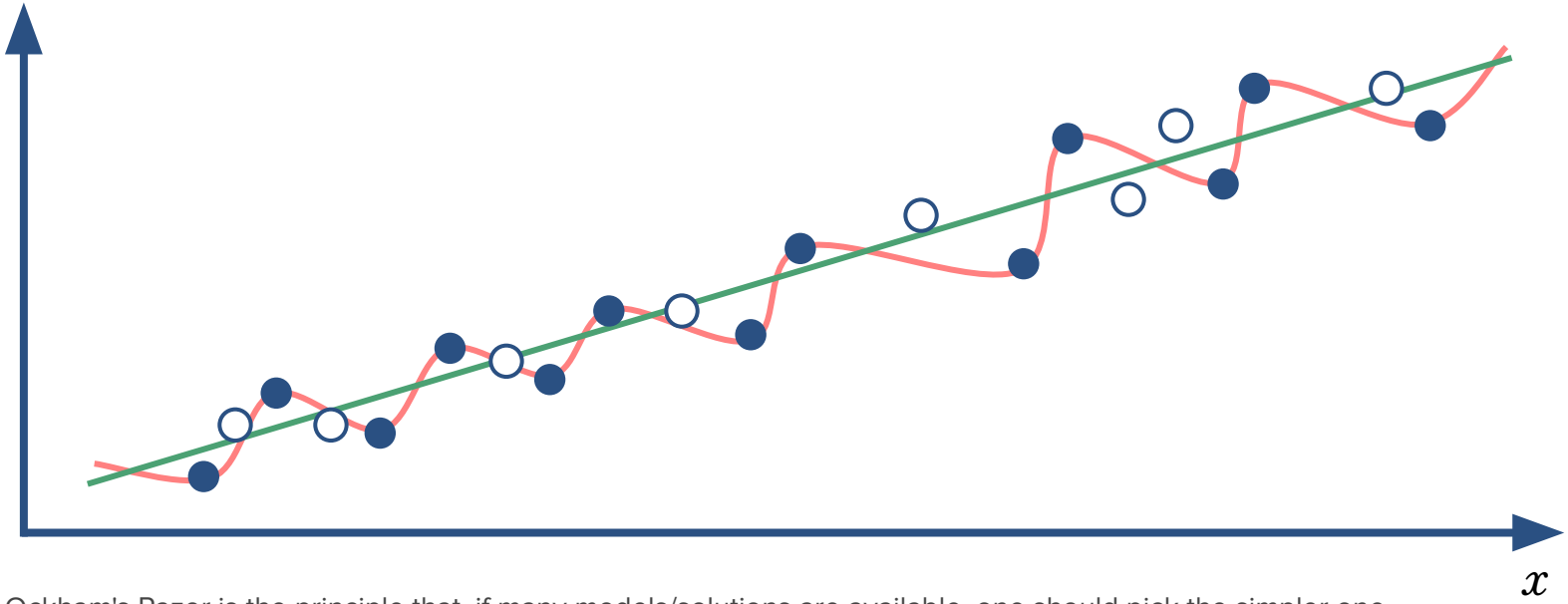
Regularization in Deep Learning

- Another one, simpler, is a line, when we realize that these points are probably noisy samples from **regressor** function.



Regularization in Deep Learning

- In general the second, simpler, option is preferred (using Ockham's Razor principle*), especially because it avoids fitting the intrinsic noise in the data (**overfitting**).



* Ockham's Razor is the principle that, if many models/solutions are available, one should pick the simpler one.

Regularization in Deep Learning

- Despite the previous example of a regression problem, an analogous example can be traced to classification tasks.
- To add regularization to Deep Learning, the usual practice is to add a **regularizer** $R(\theta)$ to its average loss function:

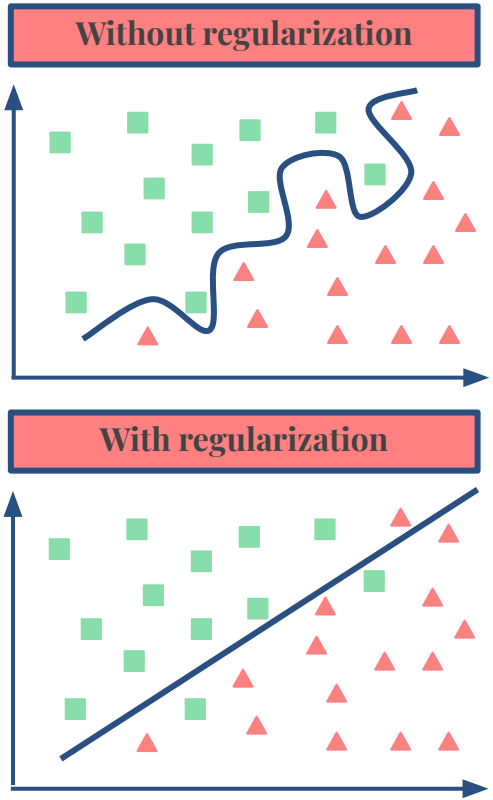
Encourages predictions to match training data

$$L(\theta) = \frac{1}{n}\sum_{i=1}^{n} l(NN_\theta(x^{(i)}), y^{(i)}) + \lambda R(\theta)$$

Prevents the model from doing too well on training data

where λ ("lambda") is a positive constant.

- In other words, $R(\theta)$ should ensure that the the network weights W_0, W_1, ..., W_L are "well-behaved".



Regularization in Deep Learning

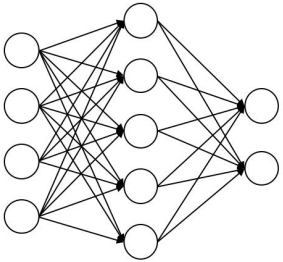
- Typical regularizers encourage the network to keep its weights **not too large**.
- Another, quite unexpectedly, regularizer is **Dropout**, which consists in randomly ignoring certain nodes in a layer during training:

Dropout: A Simple Way to Prevent Neural Networks from Overfitting

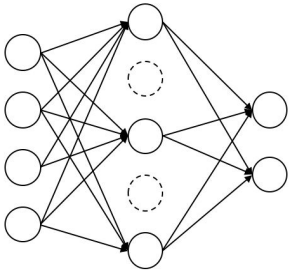
Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov, 15(56):1929–1958, 2014.

- In practice, when using dropout, we set a variable $p \in (0, 1)$ that indicates the percentage of units in a given layer will be “turn off” during training.
- At the beginning of each SGD epoch, another sample of units to turn off is randomly chosen with probability p.

Standart Network



Network With Dropout



Improving the Result: Batch-Normalization

- Another regularization technique used to improve a deep learning model is called **Batch Normalization (BN)**.
- This is similar to the rescaling phase during preprocessing: we don't want the outputs of each layer to have a very large range. Additionally, we also change their mean.
- Basically, if $x_1, x_2, ..., x_m$ are the **outputs of layer** for a given data batch, we first compute their mean and standard deviation:

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i, \ \sigma_B = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2}$$

- Then we rescale and shift each output using the following formula using the parameters γ and β, which will become the new standard deviation and mean of the x's, respectively:

$$\tilde{x}_i = \gamma\frac{(x_i - \mu_B)}{\sqrt{\sigma_B^2 + \epsilon}} + \beta$$

Improving the Result: Batch-Normalization

- This simple change was show to accelerate training and improve learning performance:

[Submitted on 11 Feb 2015 (v1), last revised 2 Mar 2015 (this version, v3)]

Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

Sergey Ioffe, Christian Szegedy

- In PyTorch, you can add it step via `nn.BatchNorm1d()` and it acts like a module in the network definition.
- On the right, we change our network to have a BN step right before the activation function of the hidden layer.

```
class MNISTNeuralNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.input_to_hidden_layer = nn.Linear(28 * 28, 1000)
        self.batch_norm = nn.BatchNorm1d(1000)
        self.hidden_layer_activation = nn.ReLU()
        self.hidden_to_output_layer = nn.Linear(1000, 10)
    def forward(self, x):
        x = self.input_to_hidden_layer(x)
        x = self.batch_norm(x)
        x = self.hidden_layer_activation(x)
        x = self.hidden_to_output_layer(x)
        return x
```

Improving the Result: Dropout

- Finally, we add a regularizer via dropout to our network, via the `nn.Dropout()` module:

```
class MNISTNeuralNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.input_to_hidden_layer = nn.Linear(28 * 28, 1000)
        self.batch_norm = nn.BatchNorm1d(1000)
        self.hidden_layer_activation = nn.ReLU()
        self.dropout = nn.Dropout(0.25)
        self.hidden_to_output_layer = nn.Linear(1000, 10)
    def forward(self, x):
        x = self.input_to_hidden_layer(x)
        x = self.batch_norm(x)
        x = self.hidden_layer_activation(x)
        x = self.dropout(x)
        x = self.hidden_to_output_layer(x)
        return x
```

In this case, we say the the weights in the following layer (`hidden_to_output_layer`) will have 25% of its units turned off randomly.

Model evaluation and training

- Having seen BN and Dropout modules, we are ready to understand what each of why we have `model.train()` and `model.eval()` in the following functions:

```
def train_batch(x, y, model, opt, loss_fn):
    model.train()
    (...)
```

```
def accuracy(x, y, model):
    model.eval()
    (...)
```

- They tell PyTorch to switch from “learning mode” to “evaluation mode”.
- This helps inform modules such as Dropout and BN, which are designed to behave differently during training and evaluation.
- For instance, in training mode, BN updates the mean of each new batch, whereas, for evaluation/testing mode, these updates do not happen.
- You can call either `model.eval()` or `model.train(mode=False)` to tell PyTorch that you are testing.

Exercise (*In pairs*)

Click here to open code in Colab

- Run the previous results on the MNIST dataset using the dropout and BN modules as shown previously, keeping all the previous parameters (such as number of epochs) the same. Keep track of learning time, final training accuracy and test accuracy. Now, remove the BN module and focus on dropout. Change p to 0.1 and then to 0.9 and note the change in train/test accuracy and learning time.
- If `x_train` has shape *[60000, 28, 28]*, which shape will it have after running `x = x.view(-1,28*28)`?

Extra Material:

- There is this very good tutorial called learnpytorch.io that helps you learn PyTorch through examples. Pretty good stuff in there.
- Tensorflow (PyTorch's competitor) has this nice website where you can train different networks in different datasets, under different parameters. It is really worth play with it

Video: *Resurrecting Abraham Lincoln with AI*

For more “resurrected” historical figures click here.