MACHINE LEARNING

Feed Forward Neural Network

Universidad Francisco de Vitoria
UFV Madrid

We move from binary classification two more complex models

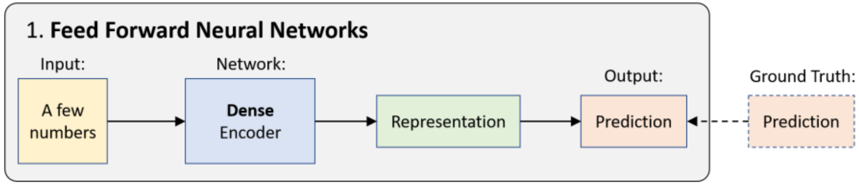
- Building a feed-forward neural network
- Loading data with Dataset and DataLoader
- Manipulating third-party mamufactured datasets
- Understanding classification loss

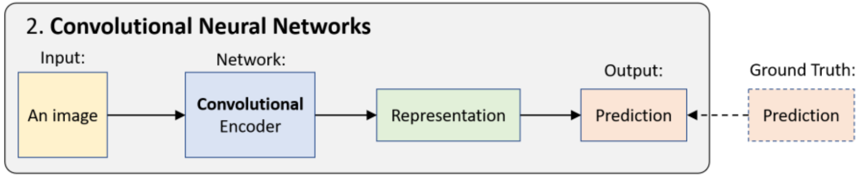
Feed Forward NN

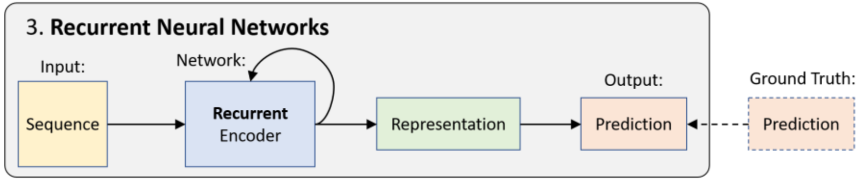
- *Building a feed-forward NN*

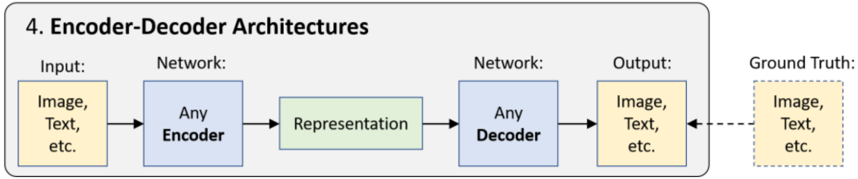
Overview

Supervised Learning

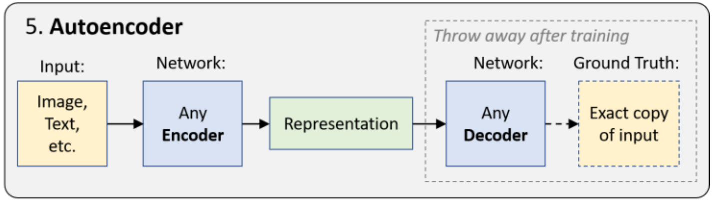


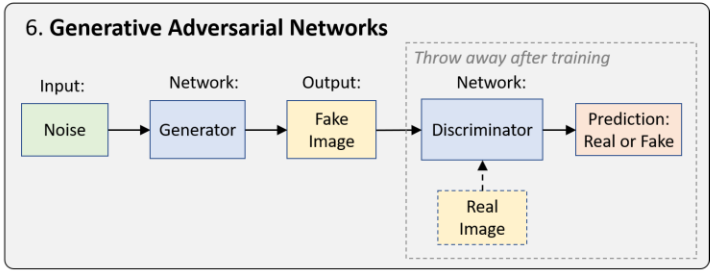




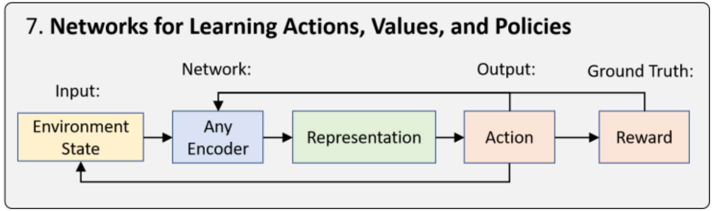


Unsupervised Learning





Reinforcement Learning



Feed forward NN

FF: Connections between the nodes do *not* form a cycle

- Each neuron in one layer has directed connections to the neurons of the subsequent layer.
- Direct flow from input to output neurons

We will see, later in the course, other types of NN that do not fit this description (recurrent and recursive NN)

We are going to develop an image classifier using a research database CIFAR-10.

CIFAR-10

CIFAR-10 consists of 60,000 tiny 32 × 32 color (RGB) images, labeled with an inte- ger corresponding to 1 of 10 classes:.

Downloading dataset

Class repository ./ml/feed_forward for code and ./ml/ data-unversioned for datasets.

Note: I will not have datasets under version control. You will have to download your own copies.

```
from torchvision import datasets

data_path = '../data/'
cifar10 = datasets.CIFAR10(data_path, train=True, download=True)
cifar10_val = datasets.CIFAR10(data_path, train=False, download=True)
```

The torchvision package, part of the PyTorch project, consists of **popular datasets**, model architectures, and common **image transformations for computer vision**.

Universidad Francisco de Vitoria
UFV Madrid

It’s key to understand how cifar-10 labels pictures

We only want to classify Birds and Airplanes.

In CIFAR10 we have 10 labels of pictures

0: Airplanes	1: Automobile
2: Bird	3: Cat
4: Deer	5: Dog
6: Frog	7: Horse
8: Ship	9: Truck

Because we are only going to classify Birds and Airplanes, we will change labels:

0 ->0 Airplanes, we keep the label

2 ->1 Birds, we change the label from 2 to 1

1 ->2 Cars, we don't need this label, we reuse the label 2 unused

Feed Forward NN

- *Building a feed-forward NN*
- *Dataset class*

The Dataset class

cifar is a subclass of the Dataset class:

```
type(cifar10).__mro__
Out[3]:
(torchvision.datasets.cifar.CIFAR10,
 torchvision.datasets.vision.VisionDataset,
 torch.utils.data.dataset.Dataset,
 typing.Generic,
 object)
```

```
 Hierarchical Method Resolution Order of CIFAR:
(<class 'torchvision.datasets.cifar.CIFAR10'>, <class
'torchvision.datasets.vision.VisionDataset'>, <class
'torch.utils.data.dataset.Dataset'>, <class 'typing.Generic'>, <class 'object'>)
```

A lot of effort in solving any machine learning problem goes into preparing the data.

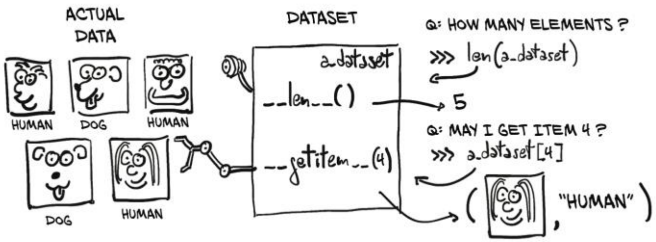
PyTorch provides many tools to make data loading.

In particular **torch.utils.data.Dataset** is an **abstract class representing a dataset.**

The Dataset class

Custom dataset should inherit Dataset and override the following methods:

- **__len__** so that len(dataset) returns the size of the dataset.
- **__getitem__** to support the indexing such that *dataset[i]* can be used to get *i-ith* sample.



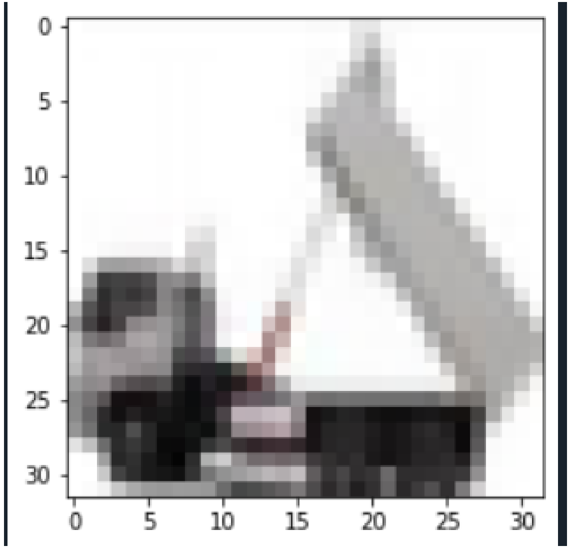

```
In [2]: len(cifar10)
Out[2]: 50000

In [3]: cifar10[50]
Out[3]: (<PIL.Image.Image image mode=RGB size=32x32>, 9)
```

data.CIFAR10 dataset implementation

Universidad Francisco de Vitoria
UFV Madrid

- __len__: with len we get the number of elements in the database, 50,000
- __getitem__: we can use the standard subscript for sequences to access individual items.
    - a PIL (Python Imaging Library, the PIL package) image
    - a label integer with the value 1, corresponding to “automobile”:

```
In [4]: img, label = cifar10[3199]
   ...: #img, label, class_names[label]
   ...:
   ...: plt.imshow(img)
   ...: plt.show()
   ...:
   ...: print("Label of the image: {0:d} \n Class name of the label:
{1:s}".format(label,class_names[label]))
Label of the image: 9
 Class name of the label: truck
```



Dataset transforms

We need tensors before we can process the dataset.

torchvision.transforms, module defines a set of composable, function-like objects that can be passed as an argument to a torchvision dataset such as datasets.CIFAR10.

- Perform transformations on the data after it is loaded but before it is returned by __getitem__.
- An example, transforms.ToTensor()

Original PIL image ranged from 0 to 255 (8 bits per channel)

- the ToTensor transform turns the data into a 32-bit floating-point per channel, scaling the values down from 0.0 to 1.0.

Dataset transforms

We need to create tensors from data and normalize data

```python
'''
Loading CIFAR10, we use three methods from torchvision Class
    Two methods from the transforms.Compose subclass:
        - ToTensor: converting Images to Torch Tensors
        - Normalize: normalize the values of the images of CIFAR10
    target_transform attribute, used to change labels

'''
cifar10 = datasets.CIFAR10(
    data_path,
    train=True,
    download=False,
    transform=transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.4915, 0.4823, 0.4468),
                             (0.2470, 0.2435, 0.2616))
    ]),
    target_transform=lambda label: label_map[label] if label in label_map else label
)


cifar10_val = datasets.CIFAR10(
    data_path,
    train=False,
    download=False,
    transform=transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.4915, 0.4823, 0.4468),
                             (0.2470, 0.2435, 0.2616))
    ]),
    target_transform=lambda label: label_map[label] if label in label_map else label
)
```

To Tensor

Normalize

Adapt labels

```python
label_map = {0: 0,
             2: 1,
             1: 2}
```

Supervised models doesn't work well with datasets with values out of the Interval (-1,1)

- It's a common practice to normalize data to values in the Interval (-1,1)
- There are Several techniques for data normalization

Z-Score

$$x' = \frac{X - \mu}{\sigma}$$

Max-Min

$$x' = \frac{x - min(x)}{max(x) - min(x)}$$

Picture

$$x' = \frac{x}{255}$$

https://medium.com/@mkc940/different-normalization-methods-a1be71fe9f1

Normalizing

Universidad Francisco de Vitoria
UFV Madrid

Stack all tensors along a n extra dimension (3)

Compute mean per channel (3, -1)

Keeps the channels

and merges the remaining dimensions

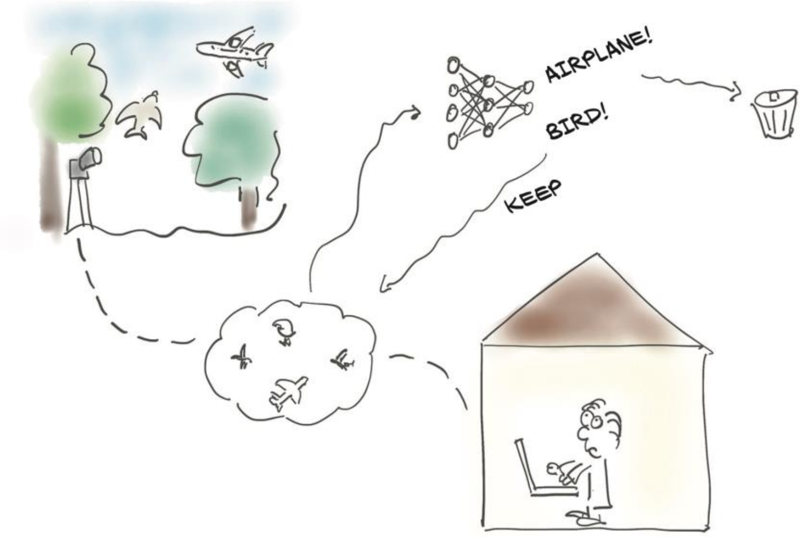
```
imgs = torch.stack([img_t for img_t, _ in tensor_cifar10], dim=3)
print("Size of the set images: {0:s}".format(str(imgs.shape)))

means = imgs.view(3, -1).mean(dim=1)
stddevs = imgs.view(3, -1).std(dim=1)


print("Mean of the values obtained in images: {0:s}".format(str(means)))

print("Variance of the values obtained in images: {0:s}".format(str(stddevs)))
```

```
IPdb [2]: !next
Mean of the values obtained in images: tensor([0.4914, 0.4822, 0.4465])

IPdb [2]: !next
Variance of the values obtained in images: tensor([0.2470, 0.2435, 0.2616])
```

Telling birds from airplanes

Subset of whole dataset, keeping only labels 0 and 2, birds and airplanes (mapped to 0, 1)

```python
label_map = {0: 0,
             2: 1,
             1: 2}

'''
Loading CIFAR10, we use three methods from torchvision Class
    Two methods from the transforms.Compose subclass:
        - ToTensor: converting Images to Torch Tensors
        - Normalize: normalize the values of the images of CIFAR10
    target_transform attribute, used to change labels

'''


cifar10 = datasets.CIFAR10(
    data_path,
    train=True,
    download=False,
    transform=transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.4915, 0.4823, 0.4468),
                             (0.2470, 0.2435, 0.2616))
    ]),
    target_transform=lambda label: label_map[label] if label in label_map else label
)

cifar10_val = datasets.CIFAR10(
    data_path,
    train=False,
    download=False,
    transform=transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.4915, 0.4823, 0.4468),
                             (0.2470, 0.2435, 0.2616))
    ]),
    target_transform=lambda label: label_map[label] if label in label_map else label
)
```



Subset method

With the class torch.utils.data.Subset(*dataset*, *indices*) it is possible to subset of a dataset at specified indices

Let's keep indices 0 and 1 (aero planes and birds).

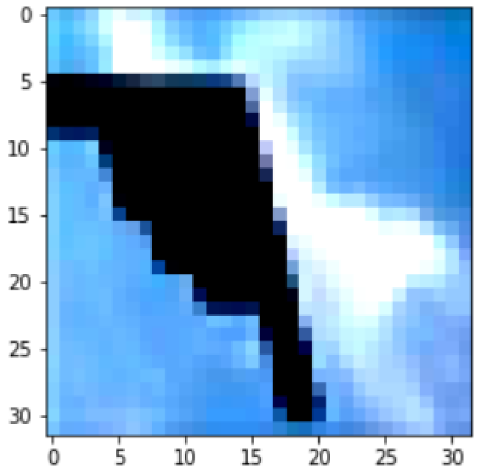
```
'''
We generate a list with the indexes of images that have 0 or 1 as label.
Remember that, CIFAR10 Class has coded a personalized __getattribute__ method that returns a tuple
The first element of the Tuple will be the Tensor with the image
The second elemento of the element will be the label
'''
birds_aeroplanes_train = [index for index, sample in enumerate(cifar10) if sample[1] in {0, 1}]
birds_aeroplanes_val = [index for index, sample in enumerate(cifar10_val) if sample[1] in {0, 1}]

'''
Filtering the tensor using the list of indexes built in the previous commands
The data Class of CIFAR has a useful method called subset, it filters the whole dataset with
the indexes that matches the labels 0 or 1 (birds or airplanes)
'''
cifar2 = torch.utils.data.Subset(cifar10, birds_aeroplanes_train)
cifar2_val = torch.utils.data.Subset(cifar10_val, birds_aeroplanes_val)
```

Other useful utilities in torch.utils: ChainDataset, ConcatDataset.

Friendly reminder

Every single picture of the Cifar10 dataset is a TUPLE

- **First element is the codified picture**
- **Second element is the label**

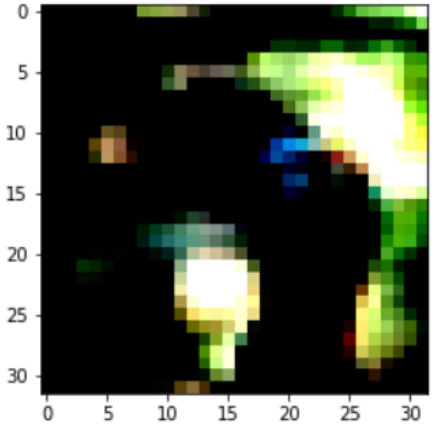


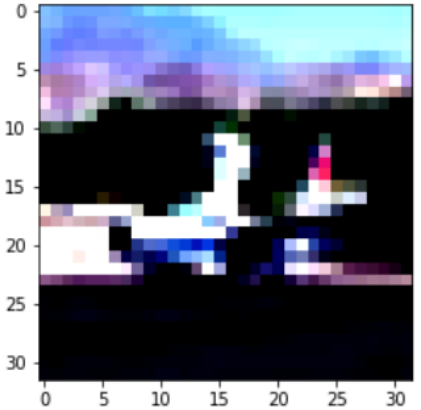
```
IPdb [5]: img_t, label = cifar2[3457]

IPdb [6]: print(label)
0

IPdb [7]: print(img_t.size())
torch.Size([3, 32, 32])

IPdb [8]: plt.imshow(img_t.permute(1, 2, 0))
```

Subset, results

class torch.utils.data.Subset(*dataset*, *indices*)

```
len(cifar2)
img_t, label = cifar2[3500]

print(label)
plt.imshow(img_t.permute(1, 2, 0))
plt.show()

len(cifar2)
img_t, label = cifar2[3501]

print(label)
plt.imshow(img_t.permute(1, 2, 0))
plt.show()
```





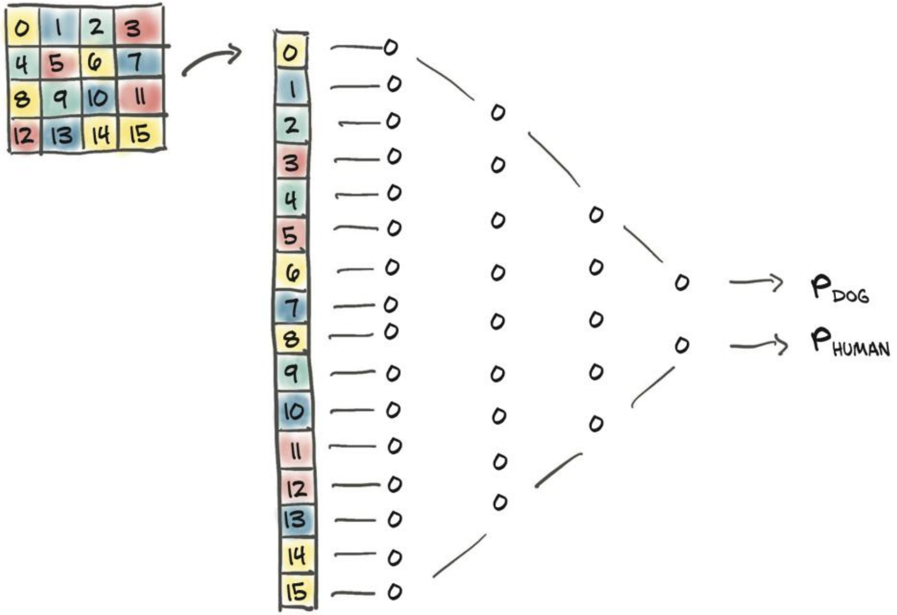
FEED FORWARD NN

- *Building a feed-forward NN*
- *Dataset class*
- *Fully connected model*

Fully connected model

Simple (naïve, guileless, artless.) approach: take the image pixels and straighten them into a long 1D vector, consider those numbers as input features

- Note: we lose spatial configuration




```
model = nn.Sequential(
    nn.Linear(3072, 512),
    nn.Tanh(),
    nn.Linear(512, 2)
    )
```

32 × 32 × 3: 3,072 input features per sample

Outputs as probabilities, Softmax

Classifier: outputs of our network, interpreted as probabilities:

- Each element of the output must be in the [0.0, 1.0] range
- The elements of the output must add up to 1.0

$$0 \leq \frac{e^{x_1}}{e^{x_1} + e^{x_2}} \leq 1$$

EACH ELEMENT BETWEEN 0 AND 1

$$\frac{e^{x_1}}{e^{x_1} + e^{x_2}} + \frac{e^{x_2}}{e^{x_1} \; e^{x_2}} = \frac{e^{x_1} \; e^{x_2}}{e^{x_1} \; e^{x_2}} = 1$$

SUM OF ELEMENTS EQUALS 1

$$\mathrm{softmax}(x_1, x_2) = \left(\frac{e^{x_1}}{e^{x_1} + e^{x_2}}, \frac{e^{x_2}}{e^{x_1} + e^{x_2}} \right)$$

$$\mathrm{softmax}(x_1, x_2, x_3) = \left(\frac{e^{x_1}}{e^{x_1} + e^{x_2} + e^{x_3}}, \frac{e^{x_2}}{e^{x_1} + e^{x_2} + e^{x_3}}, \frac{e^{x_3}}{e^{x_1} + e^{x_2} + e^{x_3}} \right)$$

$$\vdots$$

$$\mathrm{softmax}(x_1, \ldots, x_n) = \left(\frac{e^{x_1}}{e^{x_1} + \ldots + e^{x_n}}, \ldots, \frac{e^{x_n}}{e^{x_1} + \ldots + e^{x_n}} \right)$$

Softmax module (layer)

The nn module makes softmax available as a module.

- nn.Softmax requires us to specify the dimension along which the softmax function is applied

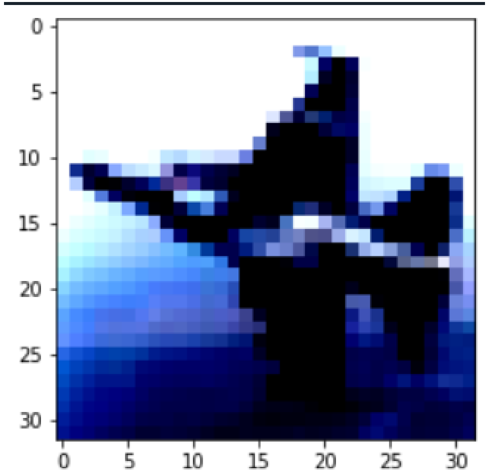
```
softmax = nn.Softmax(dim=1)
x = torch.tensor([[1.0, 2.0, 3.0],
                  [1.0, 2.0, 3.0]])
print(softmax(x))
```

```
Out[87]:
tensor([[0.0900, 0.2447, 0.6652],
        [0.0900, 0.2447, 0.6652]])
```

Model with softmax

Model with softmax:

```
model = nn.Sequential(
    nn.Linear(3072, 512),
    nn.Tanh(),
    nn.Linear(512, 2),
    nn.Softmax(dim=1)
    )
```



Checking model performance with a picture

```
img, _ = cifar2[8850]
plt.imshow(img.permute(1, 2, 0))
plt.show()
img_batch = img.view(-1).unsqueeze(0)
out = model(img_batch)
```

```
out
tensor([[0.9805, 0.0195]], grad_fn=<SoftmaxBackward0>)
```

The tensor returns the probabilities of being an airplane or a bird

Interpreting the output tensor

```
out
tensor([[0.9805, 0.0195]], grad_fn=<SoftmaxBackward0>)
```

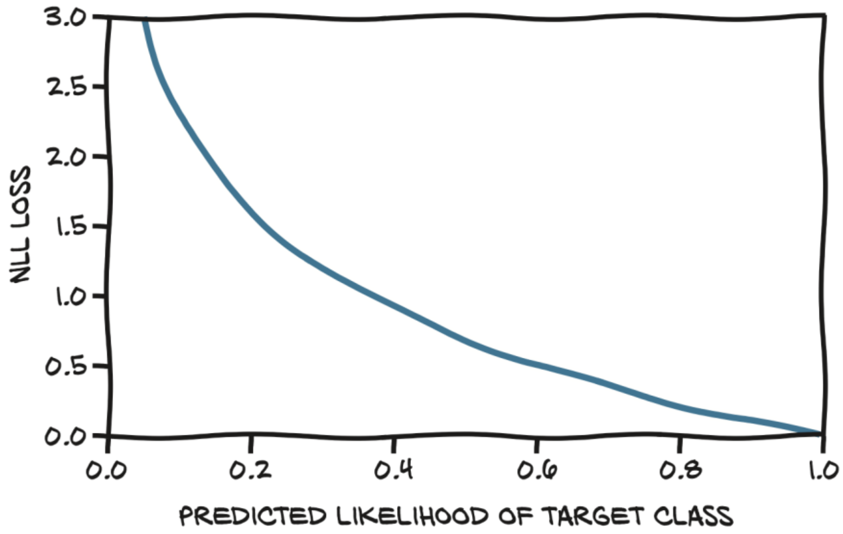
Note:

1. We have probabilities
2. _, index = torch.max(out, dim=1) gives us prediction of the label (argmax of probabilities)
3. Untrained, weights have been randomly initialized between -1 and 1
4. We have the root of the computational graph for back propagation (but we need to compute loss)

Classifier loss

As a classifier, we need to maximize *likelihood* (of our model's parameters, given the data): the probability associated with the correct class

- Loss function: negative log likelihood (NLL). NLL = - sum(log(out_i[c_i])

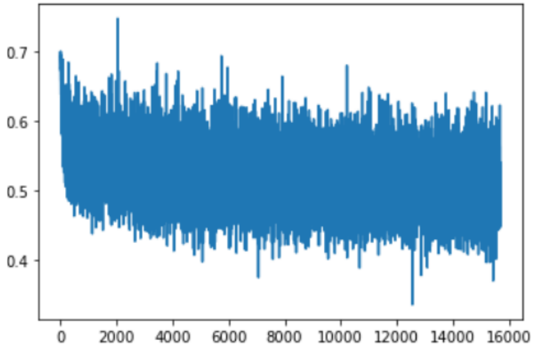


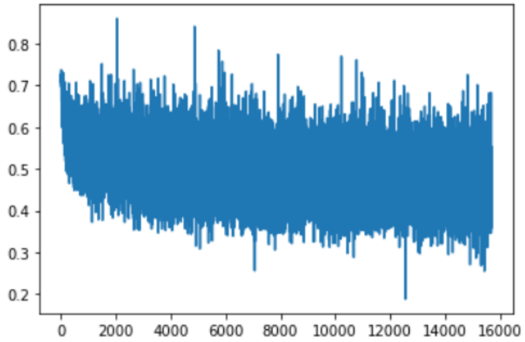
Output log probabilities

We can update from the NN log probabilities directly, log(out_i[c_i]), and then sum for samples and change sign to get the loss.

```
model = nn.Sequential(
    nn.Linear(3072, 512),
    nn.Tanh(),
    nn.Linear(512, 2),
    nn.Softmax(dim=1)
    )

learning_rate = 1e-3
```

```
model = nn.Sequential(
    nn.Linear(3072, 512),
    nn.Tanh(),
    nn.Linear(512, 2),
    nn.LogSoftmax(dim=1)
    )
```





Feed Forward NN

- *Building a feed-forward NN*
- *Dataset class*
- *Fully connected model*
- *Training the classifier*

Training the classifier

Strategies to update the model:

(A) Update with the accumulated gradient computed over all samples in the dataset

(B) Update with gradients computed with every sample

(C) Update with gradients computed over minibatches

A

```
FOR N EPOCHS:
  WITH EVERY SAMPLE IN DATASET:
    EVALUATE MODEL (FORWARD)
    COMPUTE LOSS
    ACCUMULATE GRADIENT OF LOSS
                      (BACKWARD)
  UPDATE MODEL WITH ACCUMULATED GRADIENT
```

B

```
FOR N EPOCHS:
  WITH EVERY SAMPLE IN DATASET:
    EVALUATE MODEL (FORWARD)
    COMPUTE LOSS
    COMPUTE GRADIENT OF LOSS
                      (BACKWARD)
    UPDATE MODEL WITH GRADIENT
```

C

```
FOR N EPOCHS:
  SPLIT DATASET IN MINIBATCHES
  FOR EVERY MINIBATCH:
    WITH EVERY SAMPLE IN MINIBATCH:
      EVALUATE MODEL (FORWARD)
      COMPUTE LOSS
      ACCUMULATE GRADIENT OF LOSS (BACKWARD)
    UPDATE MODEL WITH ACCUMULATED GRADIENT
```

Stochastic gradient descent

Stochastic = working on small batches (aka minibatches) of shuffled data.

- Gradients estimated over minibatches, which are poorer approximations of gradients estimated across the whole dataset, but helps convergence and prevents the optimization process from getting stuck in local minima

- Typically, minibatches are a constant size that we need to set prior to training, just like the learning rate.

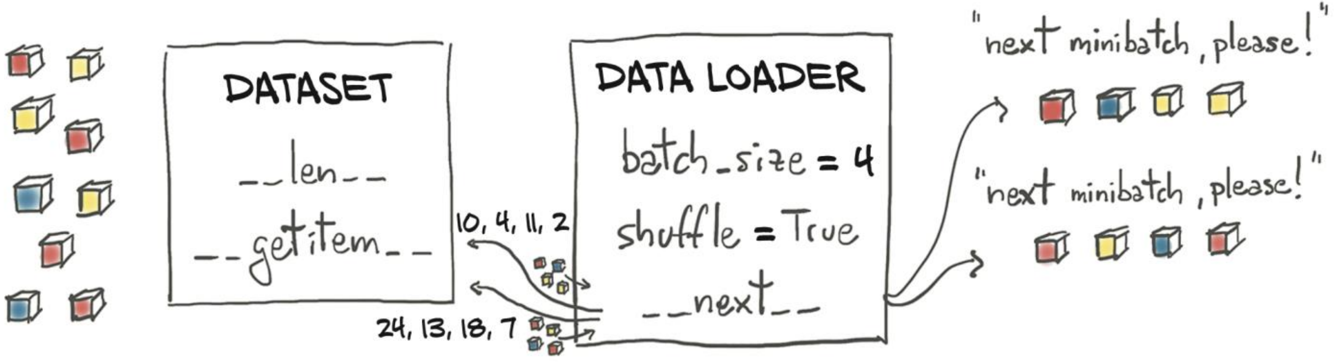
- Both batch size and learning rate are called hyperparameters, to distinguish them from the parameters of a model.

DataLoader

The torch.utils.data module has a class that helps with **shuffling and organizing the data in minibatches**: **DataLoader**.

The job of a data loader is to sample minibatches from a dataset, giving us the flexibility to choose from different sampling strategies.

- common strategy, **uniform sampling** after shuffling the data at each epoch



DataLoader

At a minimum, the DataLoader constructor takes a Dataset object as input, along with batch_size and a shuffle Boolean that indicates whether the data needs to be shuffled at the beginning of each epoch:

```
train_loader = torch.utils.data.DataLoader(cifar2, batch_size=64, shuffle=True)
```

A DataLoader can be iterated over, so we can use it directly in the inner loop of our new training code:

```
for epoch in range(n_epochs):
   for imgs, labels in train_loader:
      outputs = model(imgs.view(imgs.shape[0], -1))
      loss = loss_fn(outputs, labels)

      optimizer.zero_grad()
      loss.backward()
      optimizer.step()

   print("Epoch: %d, Loss: %f" % (epoch, float(loss)))
```

At each inner iteration, imgs is a tensor of size 64 × 3 × 32 × 32—that is, a minibatch of 64 (32 × 32) RGB image — while labels is a tensor of size 64 containing label indices.

Classifier loss

The combination of nn.LogSoftmax and nn.NLLLoss is equivalent to using nn.CrossEntropyLoss.

The nn.NLLoss computes the cross entropy but with log probability predictions as inputs where nn.CrossEntropyLoss takes scores (sometimes called *logits*).

```
loss_fn = nn.CrossEntropyLoss()
```

nn.CrossEntropyLoss

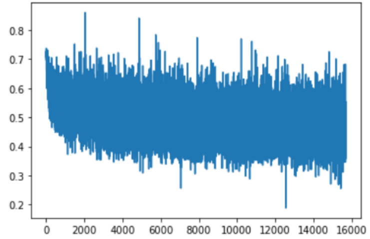
It is quite common to drop the last nn.LogSoftmax layer from the network and use nn.CrossEntropyLoss as a loss.

```
model = nn.Sequential(
    nn.Linear(3072, 512),
    nn.Tanh(),
    nn.Linear(512, 2)
    #nn.LogSoftmax(dim=1)
    )

learning_rate = 1e-4

loss_fn = nn.CrossEntropyLoss()

optimizer = optim.SGD(model.parameters(), lr=learning_rate)
```

Training accuracy (training)

Accuracy on training sample:

```
n_epochs = 100

losses=[]
for epoch in range(n_epochs):
    for imgs, labels in train_loader:
        outputs = model(imgs.view(imgs.shape[0], -1))
        loss = loss_fn(outputs, labels)
        losses.append(loss.item())
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    print("Epoch: %d, Loss: %f" % (epoch, float(loss)))
```




```
Epoch 1, Loss 0.561989
Epoch 5, Loss 0.482337
Epoch 10, Loss 0.543602
Epoch 20, Loss 0.450996
Epoch 30, Loss 0.840986
Epoch 40, Loss 0.475781
Epoch 50, Loss 0.347828
Epoch 60, Loss 0.425885
Epoch 70, Loss 0.453039
Epoch 80, Loss 0.529951
Epoch 90, Loss 0.299999
Accuracy on Training Set: 0.792800
Clipping input data to the valid range
Accuracy on Validation Set: 0.793500

In [2]:
```

Training accuracy (out of sample)

Accuracy on validation dataset:

```
val_loader = torch.utils.data.DataLoader(cifar2_val,
                                          batch_size=64,
                                          shuffle=False)

with torch.no_grad():
    for imgs, labels in val_loader:
        outputs = model(imgs.view(imgs.shape[0], -1))
        _, predicted = torch.max(outputs, dim=1)
        total += labels.shape[0]
        correct += int((predicted == labels).sum())

print("Accuracy on Validation Set: %f" % (correct / total))
```

```
Accuracy on Validation Set: 0.793500
```

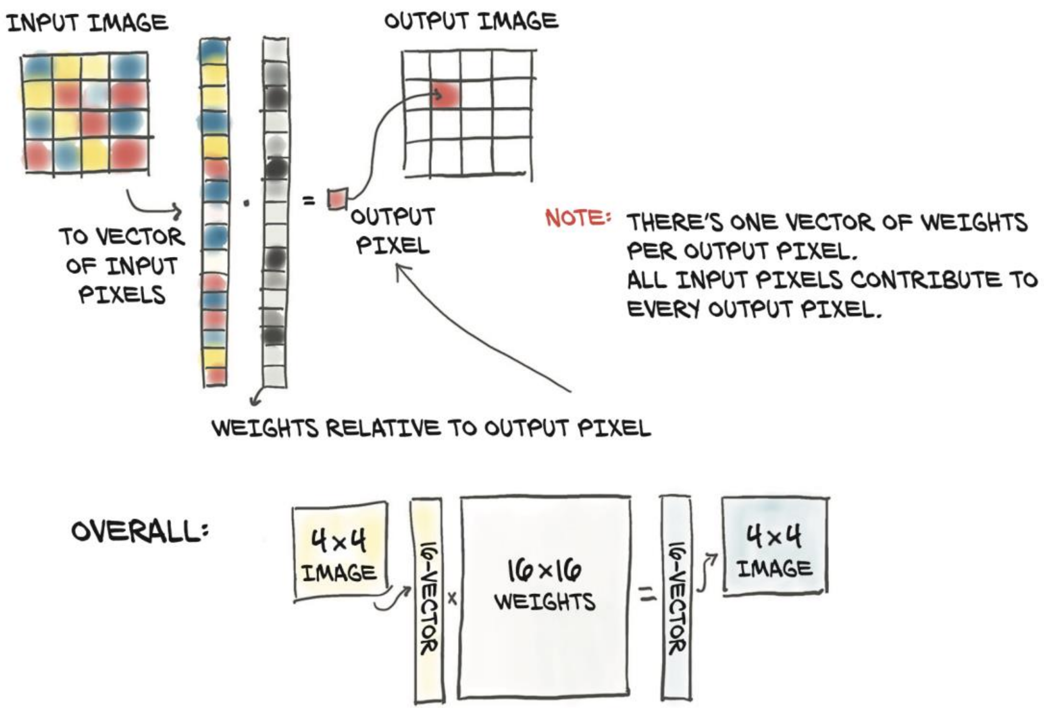
FEED FORWARD NN

- *Building a feed-forward NN*
- *Dataset class*
- *Fully connected model*
- *Training the classifier*
- *The limits of going fully connected*

The limits of going fully connected

This model takes every single input value—that is, every single component in our RGB image—and computing a linear combination of it with all the other values for every output feature.

On the other hand, we aren’t utilizing the relative position of neighboring or far away pixels, since we are treating the image as one big vector of numbers.

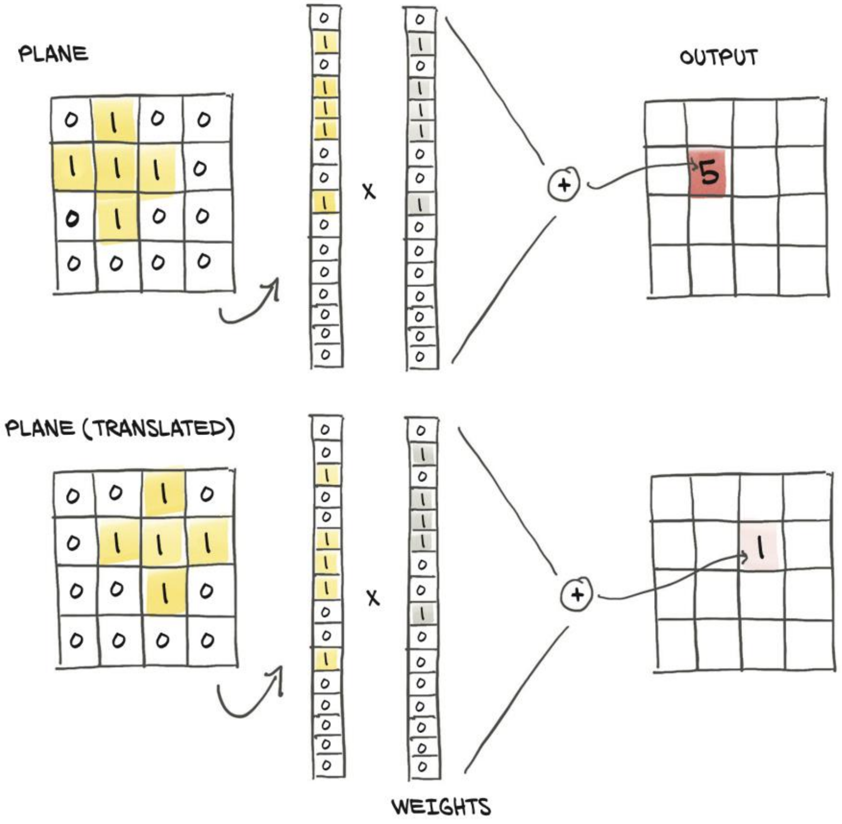


Translation invariant

Fully connected network is not *translation invariant*

If we move an image one pixel to the left the relationships between pixels will have to be relearned from scratch

We would then have to augment the dataset—that is, apply random translations to images during training— so the network would have a chance to see classify images (for every image in the dataset).



Bonus Track

There is 1 extra point in the final mark

- Current validation performance in around 79%
- Modify the model in order to achieve a validation performance over 80%

It's possible!

```
Epoch 1, Loss 0.665980
Epoch 5, Loss 0.398040
Epoch 10, Loss 0.275612
Epoch 20, Loss 0.154303
Epoch 30, Loss 0.045016
Epoch 40, Loss 0.011172
Epoch 50, Loss 0.012119
Epoch 60, Loss 0.002237
Epoch 70, Loss 0.001421
Epoch 80, Loss 0.000826
Accuracy on Training Set: 1.000000
Clipping input data to the valid range for
integers).
Accuracy on Validation Set: 0.816500
```