UNC CHARLOTTE

The WILLIAM STATES LEE COLLEGE of ENGINEERING

Introduction to ML
Lecture 16:Image Classification with Artificial Neural Networks

Hamed Tabkhi

Department of Electrical and Computer Engineering,

University of North Carolina Charlotte (UNCC)

htabkhiv@uncc.edu

Distinguishing birds from airplanes

- The problem at hand: we're going to help our friend tell birds from airplanes for her blog, by training a neural network to do the job.

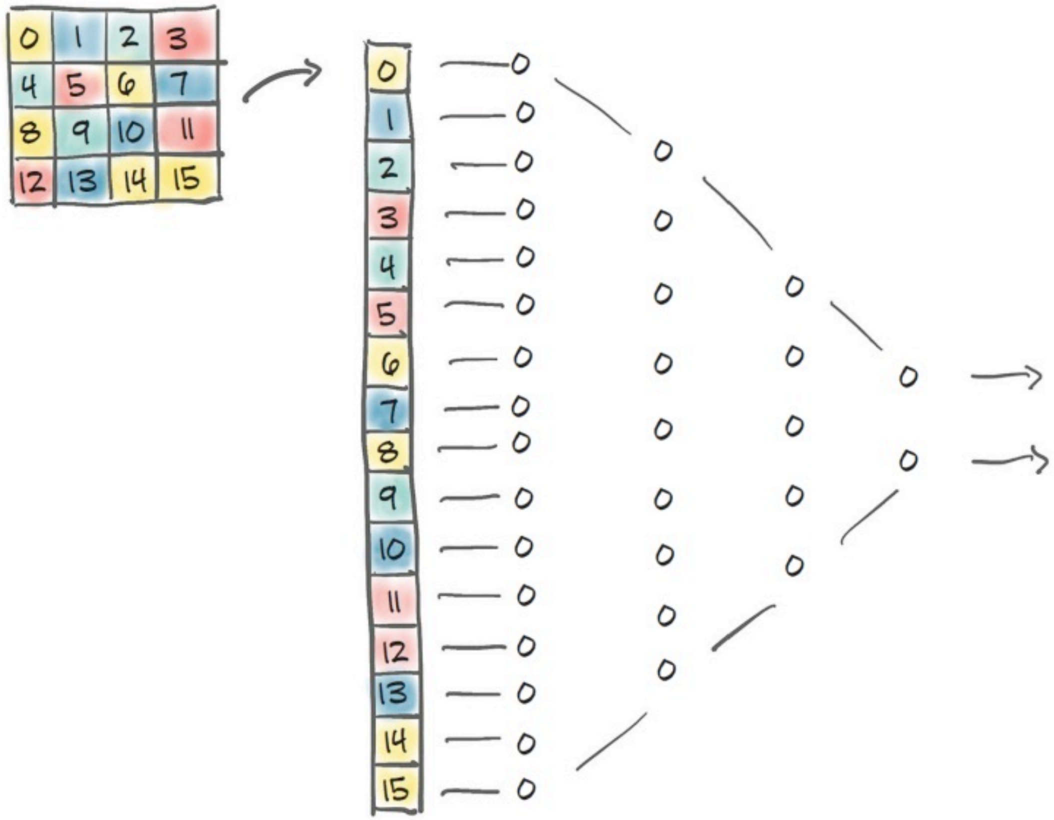
Distinguishing birds from airplanes: Building the Dataset

- We could create a `Dataset` subclass that only includes birds and airplanes.

```
# In[5]:
label_map = {0: 0, 2: 1}
class_names = ['airplane', 'bird']
cifar2 = [(img, label_map[label])
          for img, label in cifar10
          if label in [0, 2]]
cifar2_val = [(img, label_map[label])
              for img, label in cifar10_val
              if label in [0, 2]]
```

Distinguishing birds from airplanes: A fully connected model

- After all, an image is just a set of numbers laid out in a spatial configuration.
- Well, 32 × 32 × 3: that is, 3,072 input features per sample.
- Our new model would be an `nn.Linear` with 3,072 input features and some number of hidden features
- Then, followed by an activation, and then another `nn.Linear` that tapers the network down to an appropriate output number of features (2 for this use case)



Distinguishing birds from airplanes: A fully connected model

```
# In[6]:
import torch.nn as nn

n_out = 2

model = nn.Sequential(
            nn.Linear(
                3072,        # Input features
                512,         # Hidden layer size
            ),
            nn.Tanh(),
            nn.Linear(
                512,         # Hidden layer size
                n_out,       # Output classes
            )
        )
```

- We somewhat arbitrarily pick 512 hidden features.
- A neural network needs at least one hidden layer (of activations, so two modules) with a nonlinearity in between in order to be able to learn arbitrary functions

Distinguishing birds from airplanes: Output of the Classifier

- **The key realization in this case is that we can interpret our output as probabilities.**
- The first entry is the probability of “airplane,” and the second is the probability of “bird”.
- Casting the problem in terms of probabilities imposes a few extra constraints on the outputs of our network:
  1. Each element of the output must be in the `[0.0, 1.0]` range
  2. The elements of the output must add up to 1.0

Softmax

- It takes the elements of the vector, compute the elementwise exponential, and divide each element by the sum of exponentials.
- Softmax is a monotone function, in that lower values in the input will correspond to lower values in the output.
- However, it's not *scale invariant*, in that the ratio between values is not preserved.

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

EACH ELEMENT BETWEEN 0 AND 1

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

SUM OF ELEMENTS EQUALS 1

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

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

$$\vdots$$

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

Softmax

```
# In[7]:
def softmax(x):
    return torch.exp(x) / torch.exp(x).sum()
# In[8]:
x = torch.tensor([1.0, 2.0, 3.0])

softmax(x)

# Out[8]:
tensor([0.0900, 0.2447, 0.6652])

# In[9]:
softmax(x).sum()

# Out[9]:
tensor(1.)
```

- `nn.Softmax` requires us to specify the dimension along which the softmax function is applied.
- In this case, we have two input vectors in two rows, so we initialize `nn.Softmax` to operate along dimension 1.

```
# In[11]:
model = nn.Sequential(
            nn.Linear(3072, 512),
            nn.Tanh(),
            nn.Linear(512, 2),
            nn.Softmax(dim=1))
```

Negative log likelihood (NLL)

- NLL = - sum(log(out_i[c_i]))
- where the sum is taken over only the correct class for each sample in the batch (c_i)
- NLL takes probabilities as input; so, as the likelihood grows, the other probabilities will necessarily decrease

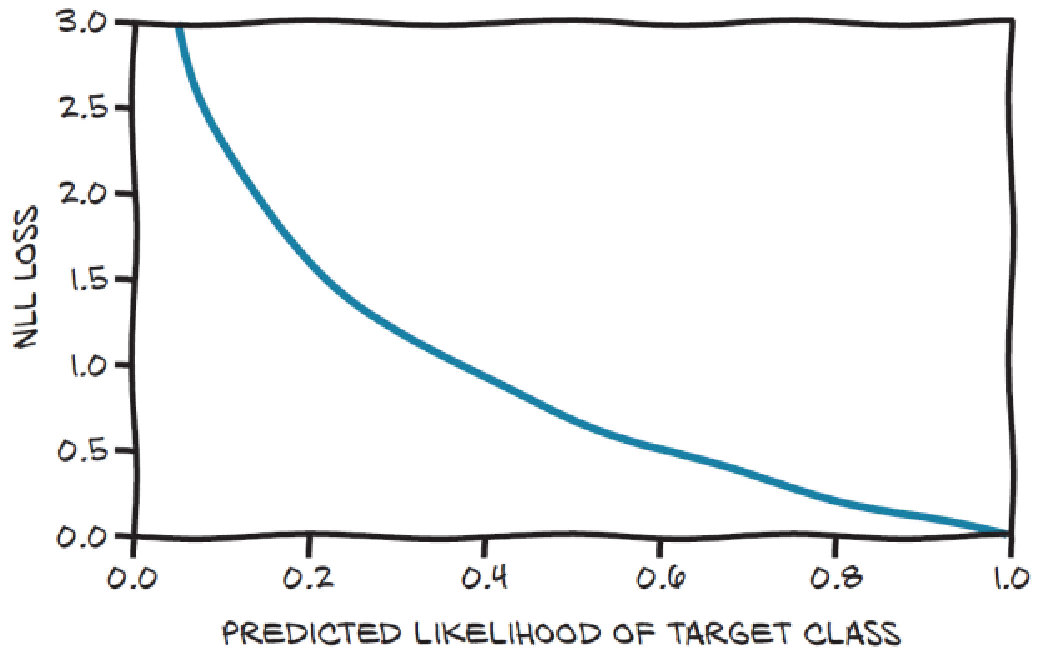
Summing up, our loss for classification can be computed as follows. For each sample in the batch:

1 Run the forward pass, and obtain the output values from the last (linear) layer.

2 Compute their softmax, and obtain probabilities.

3 Take the predicted probability corresponding to the correct class (the likelihood of the parameters). Note that we know what the correct class is as we have our ground truth.

4 Compute its logarithm, slap a minus sign in front of it, and add it to the loss.



Negative log likelihood (NLL)

- PyTorch has an `nn.NLLLoss` class.
- It does not take probabilities but rather takes a tensor of log probabilities as input.
- It then computes the NLL of our model given the batch of data.
- Taking the logarithm of a probability is tricky when the probability gets close to zero.
- The Workaround is to use `nn.LogSoftmax`, which takes care to make the calculation numerically stable.

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

img, label = cifar2[0]

out = model(img.view(-1).unsqueeze(0))

loss(out, torch.tensor([label]))
```

- The loss takes the output of `nn.LogSoftmax` for a batch as the first argument and a tensor of class indices (zeros and ones, in our case) as the second argument.

Training the Classifier

```
import torch
import torch.nn as nn

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

learning_rate = 1e-2

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

loss_fn = nn.NLLLoss()

n_epochs = 100

for epoch in range(n_epochs):
    for img, label in cifar2:
        out = model(img.view(-1).unsqueeze(0))
        loss = loss_fn(out, torch.tensor([label]))

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

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

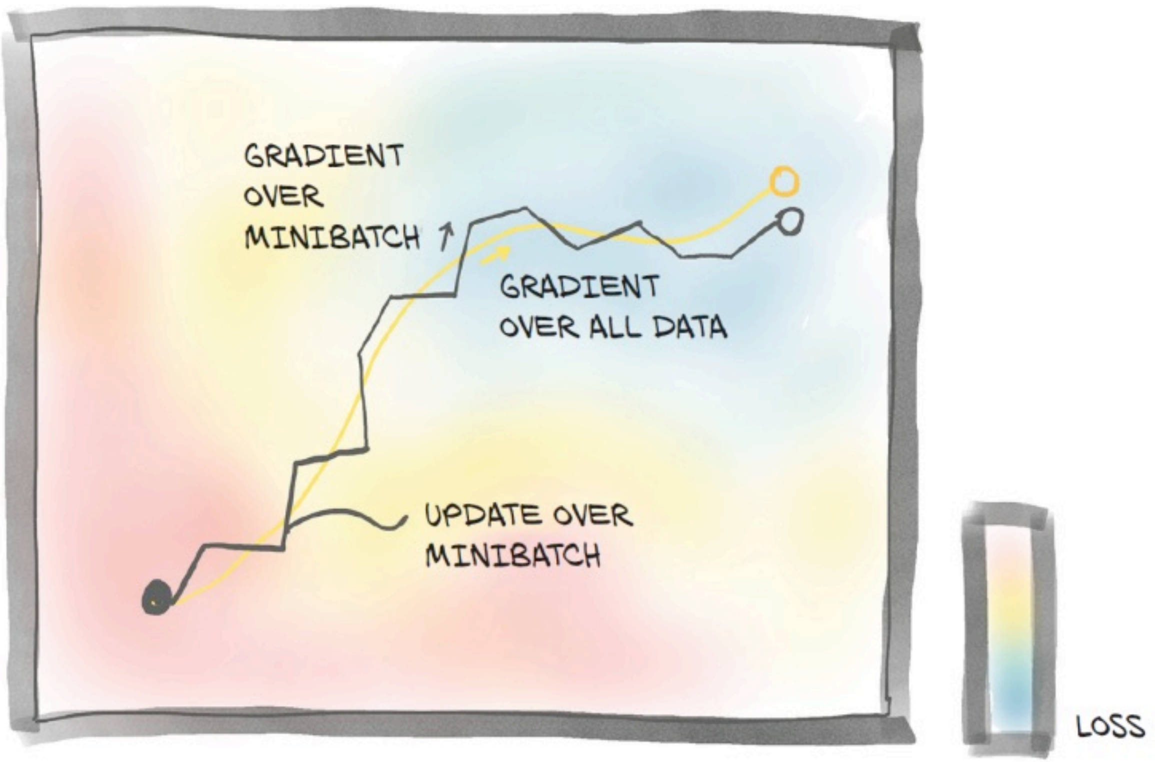
Prints the loss for the last image. In the next chapter, we will improve our output to give an average over the entire epoch.

Averaging updates over minibatches

- Evaluating all 10,000 images in a single batch would be too much, so we decided to have an inner loop where we evaluate one sample at a time and backpropagate over that single sample.
- we apply changes to parameters based on a very partial estimation of the gradient on a single sample.
- However, what is a good direction for reducing the loss based on one sample might not be a good direction for others.
- By shuffling samples at each epoch and estimating the gradient on one or (preferably, for stability) a few samples at a time, we are effectively introducing randomness in our gradient descent.
- It stands for *stochastic gradient descent, and S* is about working on small batches (minibatches) of shuffled data.

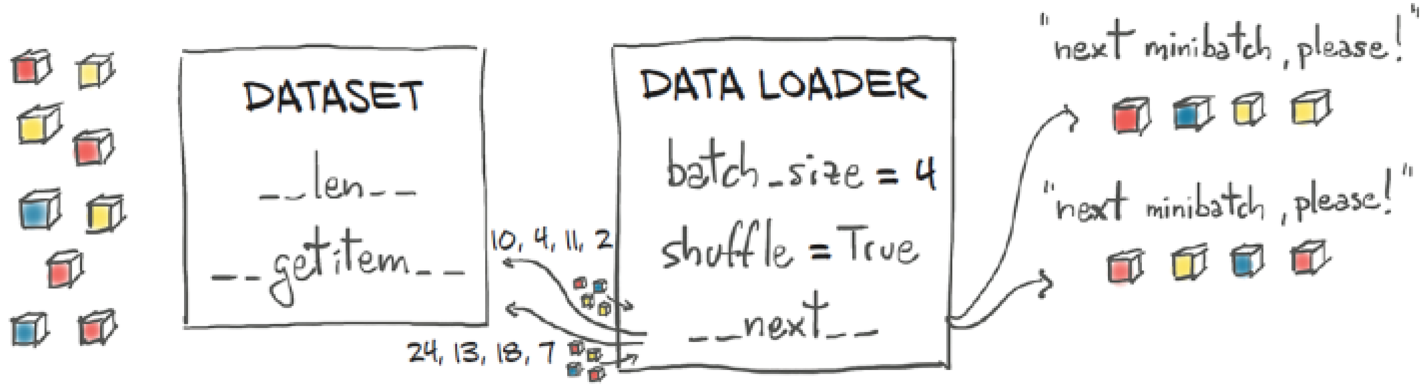
Averaging updates over minibatches

- Yellow Path: Gradient descent averaged over the whole dataset
- **Dark Path:** Stochastic gradient descent, where the gradient is estimated on randomly picked minibatches
- gradients from minibatches are randomly off the ideal trajectory, which is part of the reason why we want to use a reasonably small learning rate.
- Shuffling the dataset at each epoch helps ensure that the sequence of gradients estimated over minibatches is representative of the gradients computed across the full dataset.
- Typically, minibatches are a constant size that we need to set prior to training, just like the learning rate. These are called ***hyperparameters**.*



Data Loader

- In our initial training code, we chose minibatches of size 1 by picking one item at a time from the dataset.
- The `torch.utils.data` module has a class that helps with shuffling and organizing the data in minibatches, which we call it **`DataLoader`**.
- The job of a data loader is to sample minibatches from a dataset, giving us the flexibility to choose from different sampling strategies.
- A very common strategy is uniform sampling after shuffling the data at each epoch.




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

Putting Everything Together

```
import torch
import torch.nn as nn

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


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

learning_rate = 1e-2

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

loss_fn = nn.NLLLoss()

n_epochs = 100

for epoch in range(n_epochs):
    for imgs, labels in train_loader:

        batch_size = imgs.shape[0]
        outputs = model(imgs.view(batch_size, -1))
        loss = loss_fn(outputs, labels)

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

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

Due to the shuffling, this now prints the loss for a random batch—clearly something we want to improve in chapter 8.

- At each inner iteration, `imgs` is a tensor of size 64 × 3 × 32 × 32—that is, a minibatch of 64 (32 × 32) RGB images.

Accuracy Measurement

- We see that the loss decreases somehow, but we have no idea whether it's low enough.

```
Epoch: 0, Loss: 0.523478
Epoch: 1, Loss: 0.391083
Epoch: 2, Loss: 0.407412
Epoch: 3, Loss: 0.364203
...
Epoch: 96, Loss: 0.019537
Epoch: 97, Loss: 0.008973
Epoch: 98, Loss: 0.002607
Epoch: 99, Loss: 0.026200
```

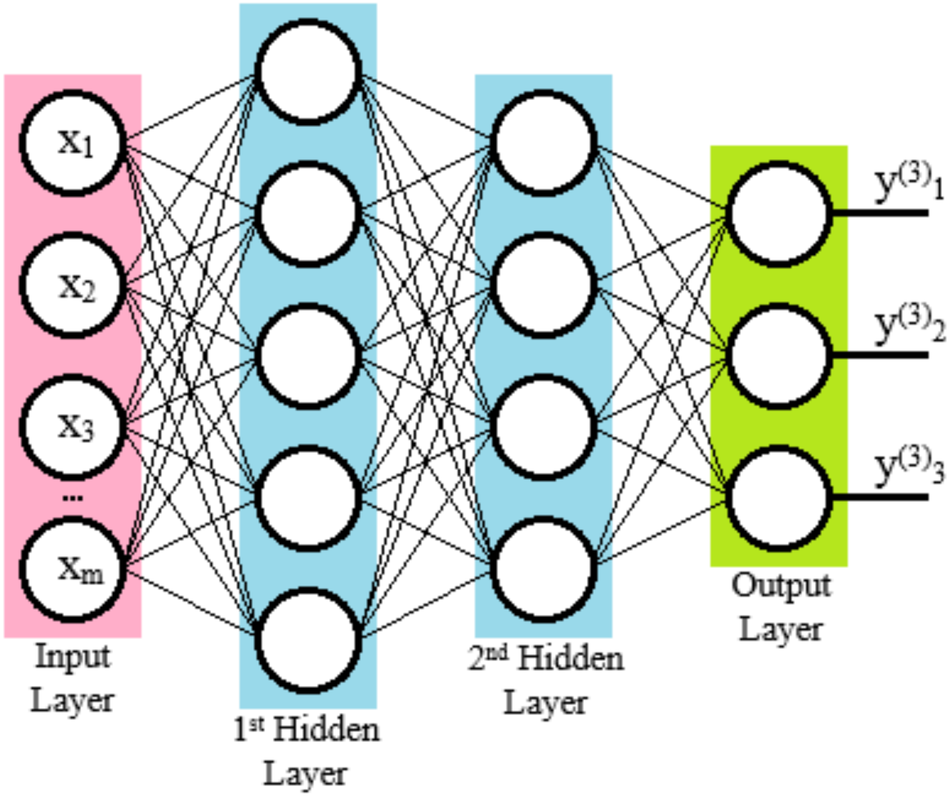
- Our model was quite a shallow classifier; it's a miracle that it worked at all.
- It did because our dataset is really simple—a lot of the samples in the two classes likely have systematic differences that help the model tell birds from airplanes, based on a few pixels.

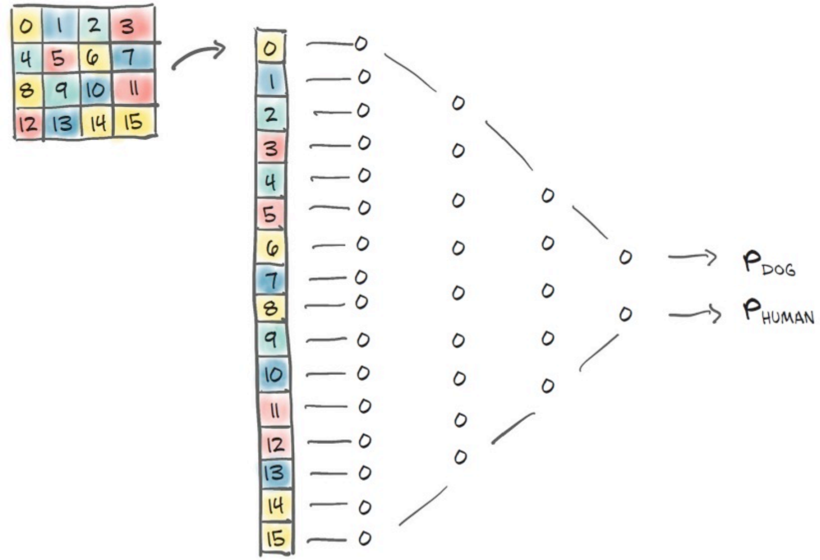
- We can compute the accuracy of our model on the validation, set in terms of the number of correct classifications over the total:

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

correct = 0
total = 0

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

print("Accuracy: %f", correct / total)
```

```
Accuracy: 0.794000
```

Fully Connected Neural Networks





The WILLIAM STATES LEE COLLEGE of ENGINEERING
UNC CHARLOTTE

Increasing Model Complexity:
Hidden Fully Connected Layers

```
model = nn.Sequential(
            nn.Linear(3072, 1024),
            nn.Tanh(),
            nn.Linear(1024, 512),
            nn.Tanh(),
            nn.Linear(512, 128),
            nn.Tanh(),
            nn.Linear(128, 2),
            nn.LogSoftmax(dim=1))
```

- Intermediate layers will do a better job of squeezing information in increasingly shorter intermediate outputs.

- It is quite common to drop the last `nn.LogSoftmax` layer from the network and use `nn.CrossEntropyLoss` as a loss.
- The combination of `nn.LogSoftmax` and `nn.NLLLoss` is equivalent to using `nn.CrossEntropyLoss`.

```
model = nn.Sequential(
            nn.Linear(3072, 1024),
            nn.Tanh(),
            nn.Linear(1024, 512),
            nn.Tanh(),
            nn.Linear(512, 128),
            nn.Tanh(),
            nn.Linear(128, 2))

loss_fn = nn.CrossEntropyLoss()
```

Increasing Model Complexity:
Hidden Fully Connected Layers

- Training accuracy: 0.998100, Validation Accuracy: 0.802000
- Our fully connected model is finding a way to discriminate birds and airplanes on the training set by memorizing the training set, but performance on the validation set is not all that great, even if we choose a larger model -> Over-fitting

Measuring Model Complexity

- PyTorch offers a quick way to determine how many parameters a model has through the `parameters()` method of `nn.Model`
- To find out how many elements are in each tensor instance, we can call the `numel` method. Summing those gives us our total count.
- By setting `requires_grad` to `True`. We might want to differentiate the number of *trainable* parameters from the overall model size.

```
# In[7]:
numel_list = [p.numel()
              for p in connected_model.parameters()
              if p.requires_grad == True]
sum(numel_list), numel_list

# Out[7]:
(3737474, [3145728, 1024, 524288, 512, 65536, 128, 256, 2])
```

Wow, 3.7 million parameters! Not a small network for such a small input image

Measuring Model Complexity

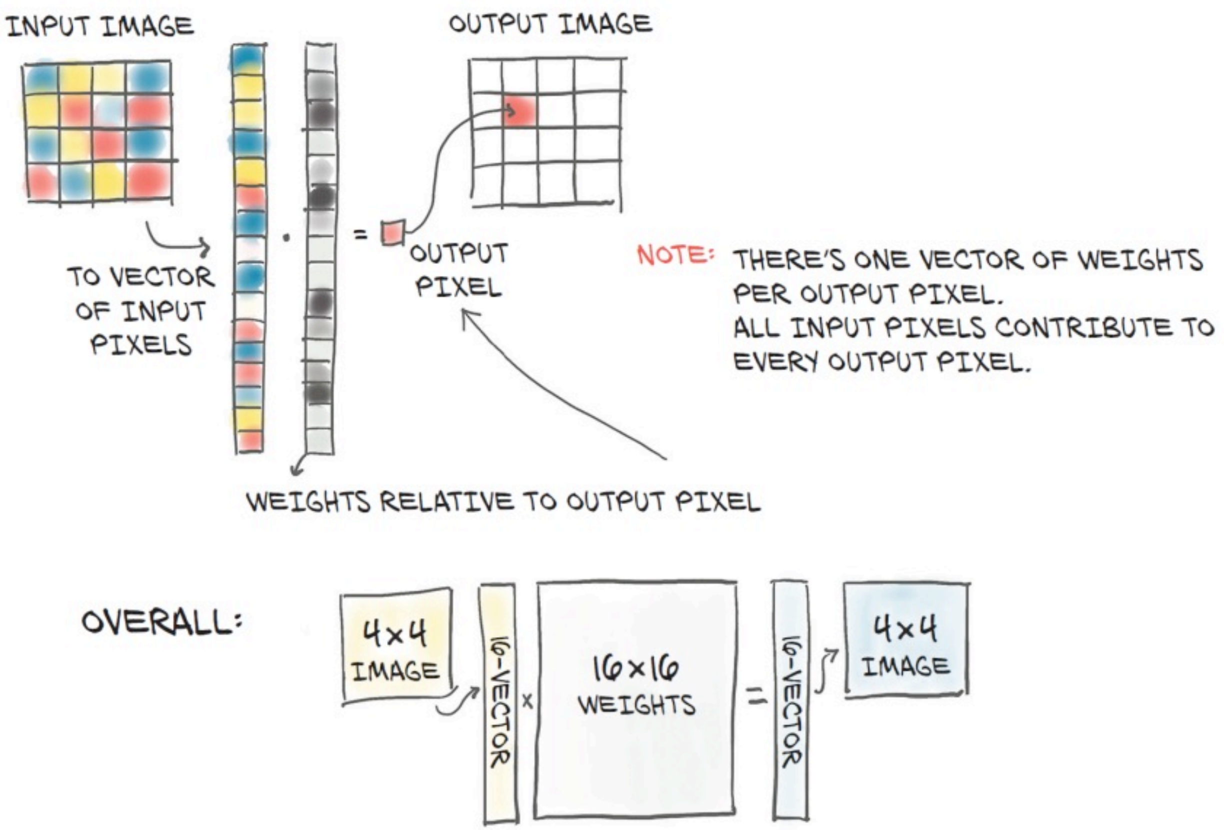
```
# In[9]:
numel_list = [p.numel() for p in first_model.parameters()]
sum(numel_list), numel_list

# Out[9]:
(1574402, [1572864, 512, 1024, 2])
```

Even our first network was pretty large, with 1.5 M parameters

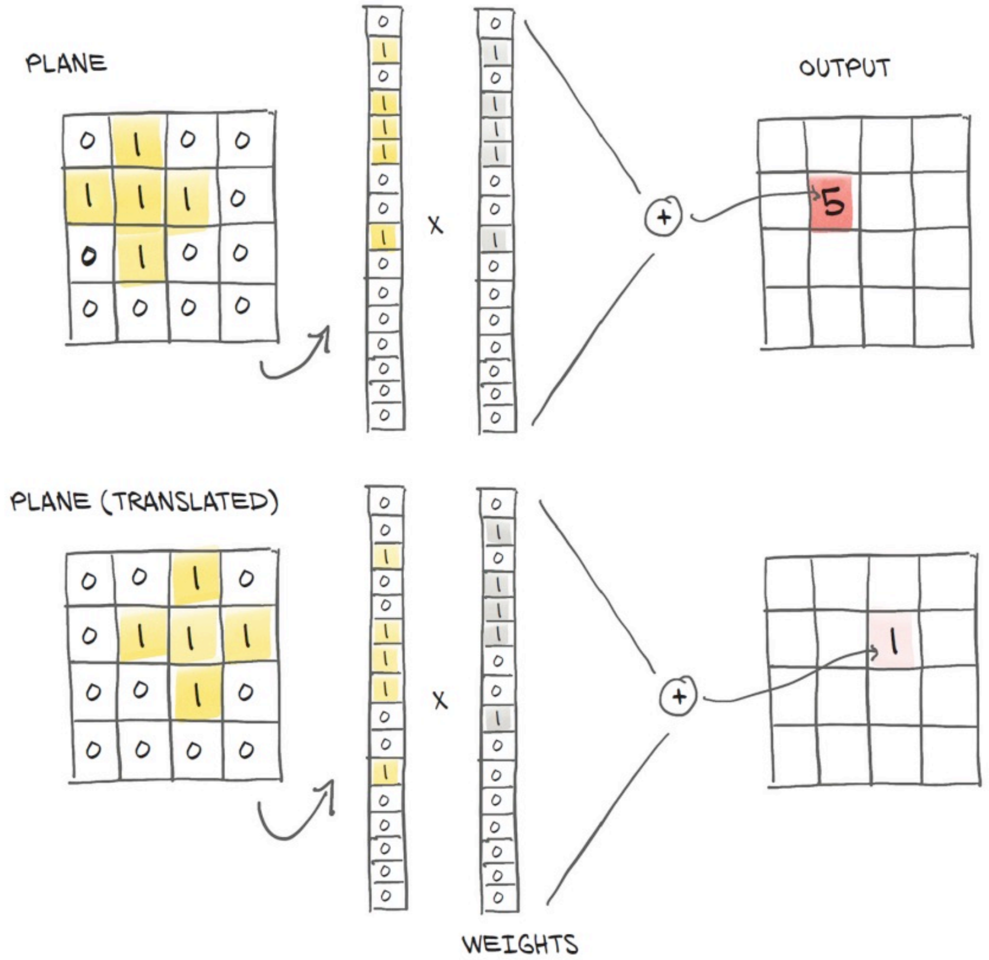
```
# In[10]:
linear = nn.Linear(3072, 1024)

linear.weight.shape, linear.bias.shape

# Out[10]:
(torch.Size([1024, 3072]), torch.Size([1024]))
```

1024 linear equations with 3072 unique weight parameters (equal to input variables) per each equation.

The Limits of Going Fully Connected



Not being *translation invariant*



- Shift the same airplane by one pixel or more as in the bottom half of the figure, and the relationships between pixels will have to be relearned from scratch.
- This time, an airplane is likely when pixel 0,2 is dark, pixel 1,2 is dark, and so on.
- In more technical terms, a fully connected network is not *translation invariant*.