UNC CHARLOTTE

The WILLIAM STATES LEE COLLEGE *of* ENGINEERING

Introduction to ML
Lecture 18: Artificial Neural Networks

Hamed Tabkhi

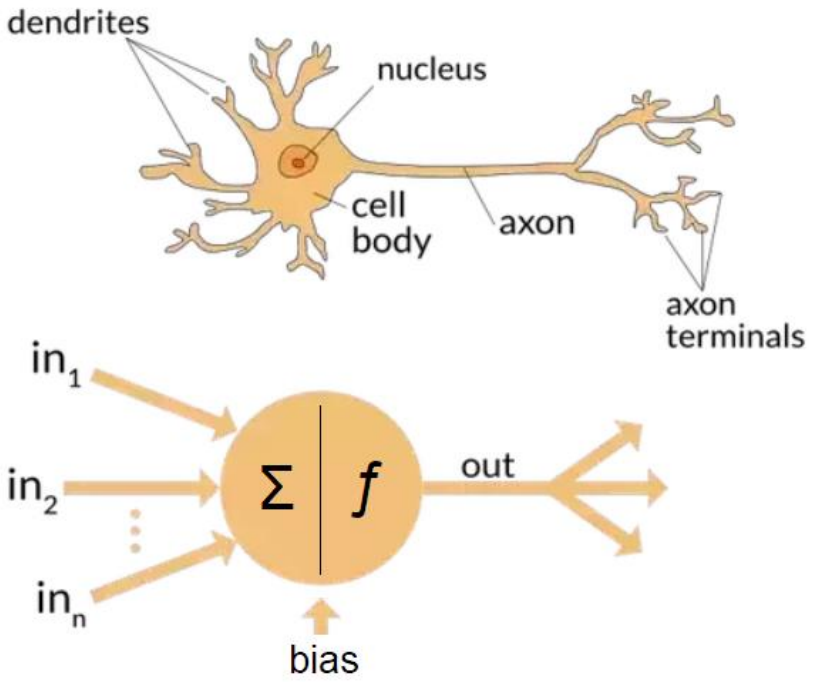
Department of Electrical and Computer Engineering,

University of North Carolina Charlotte (UNCC)

htabkhiv@uncc.edu

Neurons

- The initial models were inspired by neuroscience
- It seems likely that both artificial and physiological neural networks use vaguely similar mathematical strategies for approximating complicated functions because that family of strategies works very effectively
- Modern artificial neural networks bear only a slight resemblance to the mechanisms of neurons
- the simplest unit in (deep) neural networks is a linear operation (scaling + offset) followed by an activation function



Mathematical Expression of Neuron

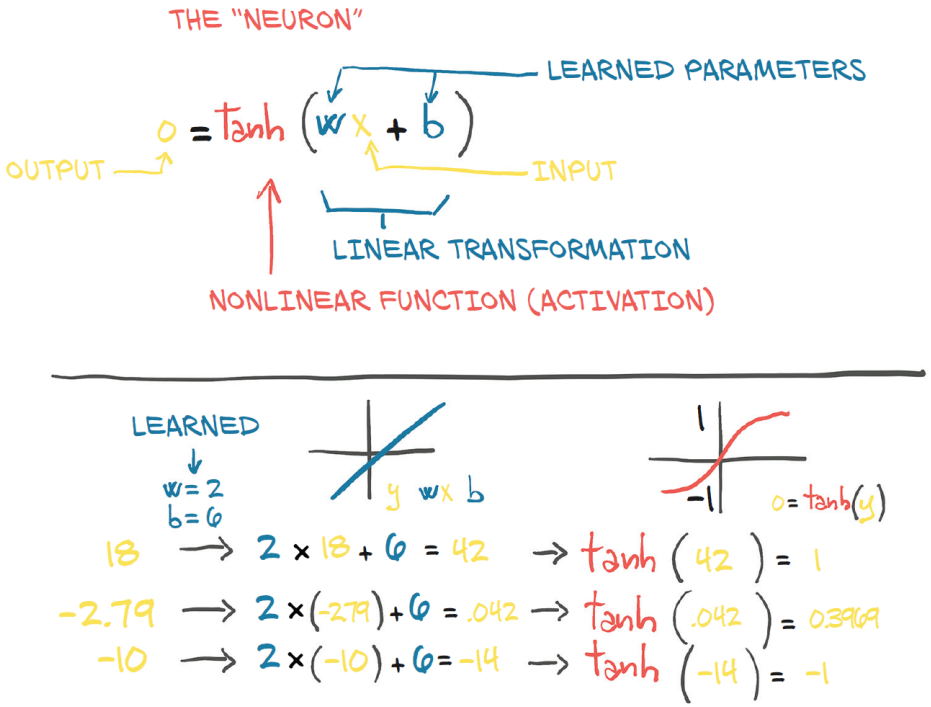
- At its core, it is nothing but a linear transformation of the input.
- Neuron multiplies the input by a number [the *weight*] and adding a constant [the *bias*]

- It then followed by the application of a fixed nonlinear function - referred to as the *activation function*

- Mathematically, we can write this out as

$$o = f(w * x + b)$$

with x as our input, w our weight or scaling factor, and b as our bias or offset. f is our activation function, set to the hyperbolic tangent, or tanh function here.

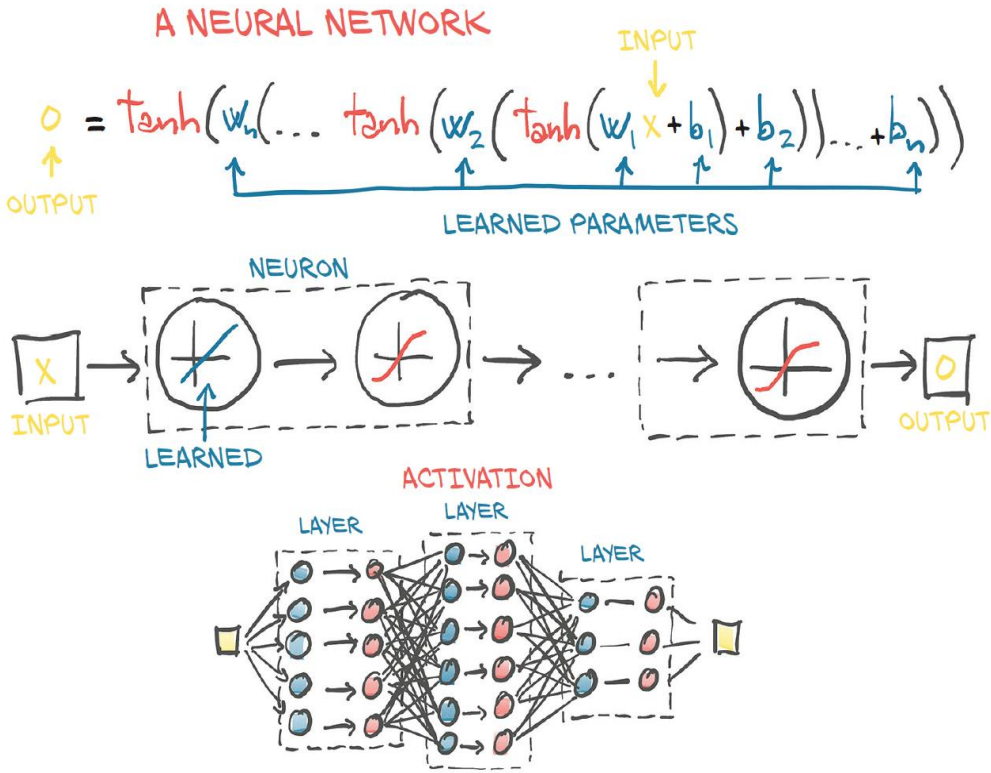


layer of neurons

It represents many neurons via the multidimensional weights and biases.

```
x_1 = f(w_0 * x + b_0)
x_2 = f(w_1 * x_1 + b_1)
...
y = f(w_n * x_n + b_n)
```

- The output of a layer of neurons is used as an input for the following layer.
- Remember that `w_0` here is a matrix, and `x` is a vector!
- Using a vector allows `w_0` to hold an entire *layer* of neurons, not just a single weight



Activation Function

- Activation functions are nonlinear. Repeated applications of (`w*x+b`) without an activation function results in a function of the same (affine linear) form.
- The nonlinearity allows the overall network to approximate more complex functions.
- Without these characteristics, the network either falls back to being a linear model or becomes difficult to train.
- Activation function is differentiable, so that gradients can be computed through them.
- The ability of an ensemble of neurons to approximate a very wide range of useful functions depends on the combination of the linear and nonlinear behavior inherent to each neuron.

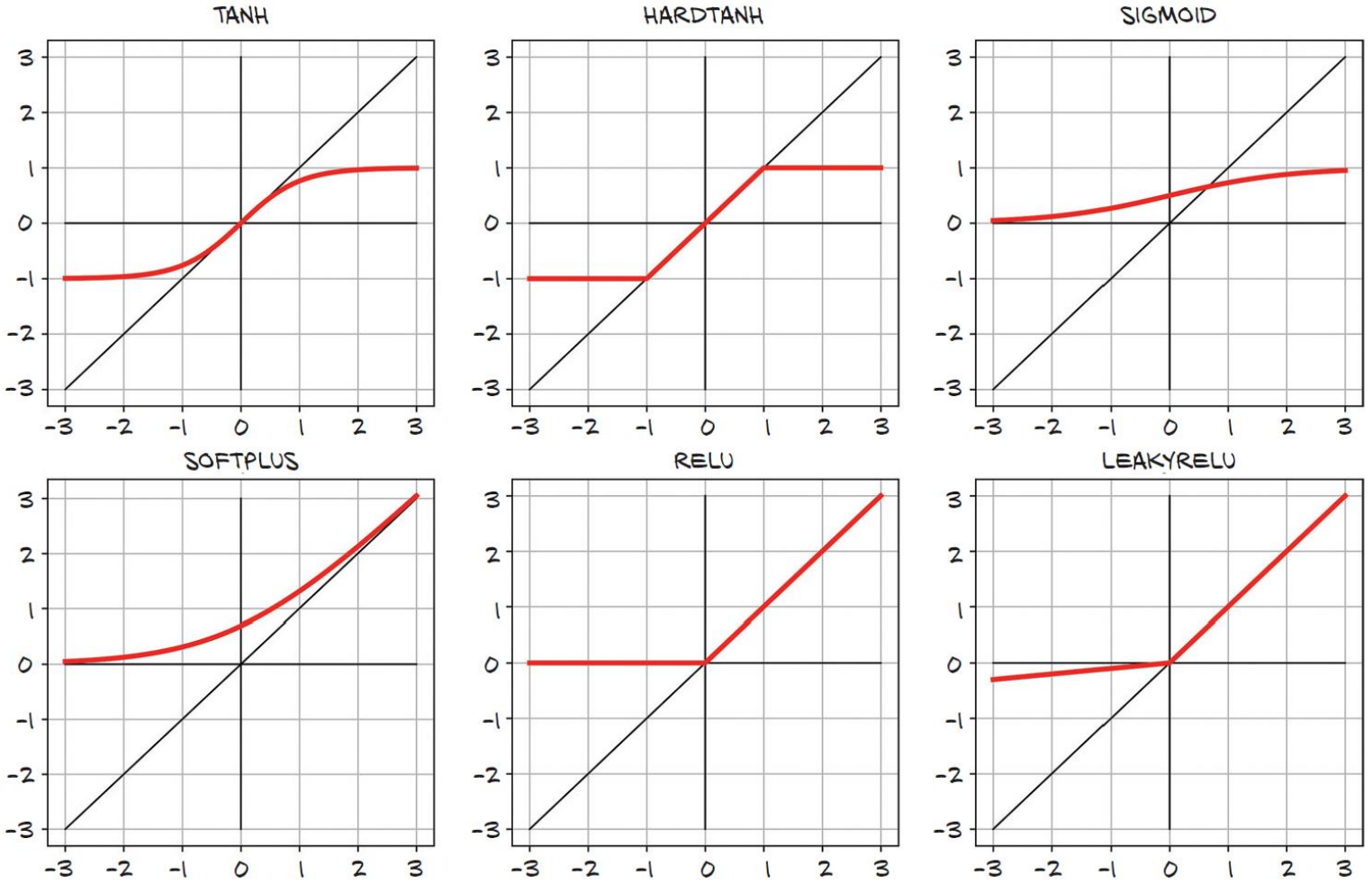
The activation function plays two important roles:

1- In the inner parts of the model, it allows the output function to have different slopes at different values—something a linear function by definition cannot do.

- By trickily composing these differently sloped parts for many outputs, neural networks can approximate arbitrary functions.

2- At the last layer of the network, it has the role of concentrating the outputs of the preceding linear operation into a given range.

Different Activation Functions



The most popular activation functions

- `ReLU` (for *rectified linear unit*) deserves special note, as it is currently considered one of the best-performing general activation functions; many state-of-the-art results have used it.
- The `Sigmoid` activation function, also known as the *logistic function*, was widely used in early deep learning work but has since fallen out of common use except where we explicitly want to move to the 0…1 range: for example, when the output should be a probability.
- `LeakyReLU` function modifies the standard `ReLU` to have a small positive slope, rather than being strictly zero for negative inputs (typically this slope is 0.01, but it's shown here with slope 0.1 for clarity).

Training of Neural Networks
(Many layers of Linear + Activation functions)

- **We are starting to get a deeper intuition for how joining many linear + activation units in parallel and stacking them one after the other leads us to a mathematical object that is capable of approximating complicated functions. Different combinations of units will respond to inputs in different ranges, and those parameters for those units are relatively easy to optimize through gradient descent, since learning will behave a lot like that of a linear function until the output saturates.**
- Thinking of what we know about how backpropagation works, we can figure out that the errors will propagate backward through the activation more effectively when the inputs are in the response range, while errors will not greatly affect neurons for which the input is saturated.
- In a network built out of linear + activation units, when different inputs are presented to the network, (a) different units will respond in different ranges for the same inputs, and (b) the errors associated with those inputs will primarily affect the neurons operating in the sensitive range, leaving other units more or less unaffected by the learning process.

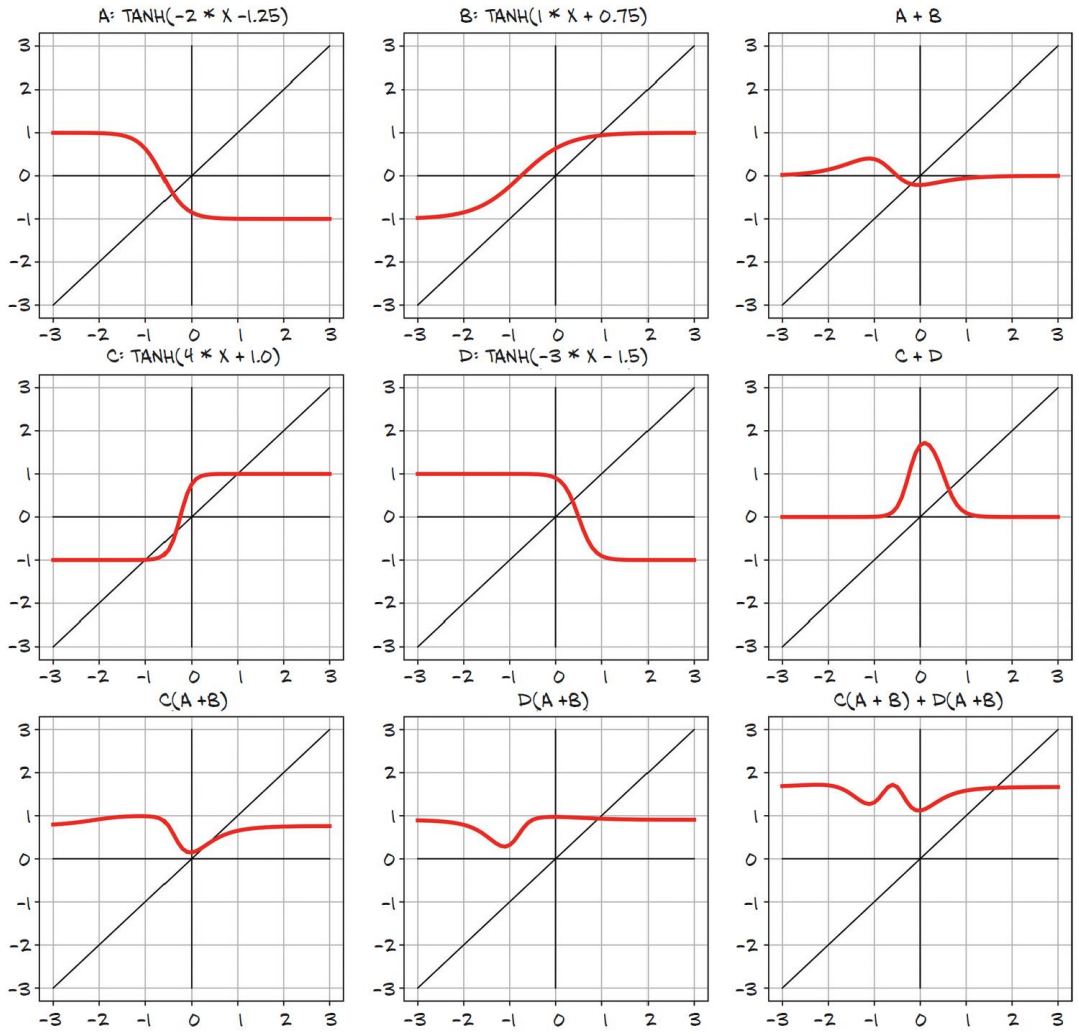
What learning means for a neural network

- Building models out of stacks of linear transformations followed by differentiable activations leads to models that can approximate highly nonlinear processes
- Deep neural networks give us the ability to approximate highly nonlinear phenomena without having an explicit model for them.
- What makes using deep neural networks so attractive is that it saves us from worrying too much about the exact function that represents our data—whether it is quadratic, piecewise polynomial, or something else.
- With a deep neural network model, we have a universal approximator and a method to estimate its parameters.
- This approximator can be customized to our needs, in terms of model capacity and its ability to model complicated input/output relationships, just by composing simple building blocks.

Nature of Error in Neural Networks

- Neural networks do not have that same property of a convex error surface, even when using the same error-squared loss function!
- There's no single right answer for each parameter we're attempting to approximate. Instead, we are trying to get all of the parameters, when acting *in concert*, to produce a useful output.
- Since that useful output is only going to *approximate* the truth, there will be some level of imperfection.
- Where and how imperfections manifest is somewhat arbitrary, and by implication the parameters that control the output (and, hence, the imperfections) are somewhat arbitrary as well.

What learning means for a neural network



- Training consists of finding acceptable values for these weights and biases so that the resulting network correctly carries out a task.

PyTorch nn Module

- PyTorch has a whole submodule dedicated to neural networks, called `torch.nn`.
- It contains the building blocks (parlance / *layers)* needed to create all sorts of neural network architectures.

```
# In[5]:
import torch.nn as nn

linear_model = nn.Linear(1, 1)
linear_model(t_un_val)

# Out[5]:
tensor([[0.6018],
        [0.2877]], grad_fn=<AddmmBackward>)

# In[6]:
linear_model.weight

# Out[6]:
Parameter containing:
tensor([[-0.0674]], requires_grad=True)

# In[7]:
linear_model.bias

# Out[7]:
Parameter containing:
tensor([0.7488], requires_grad=True)
```

We'll look into the constructor arguments in a moment.

```
y = model(x)
y = model.forward(x)
```

Correct!

Silent error. Don't do it!

Batching

- Assuming we need to run `nn.Linear` on 10 samples, we can create an input tensor of size $B \times Nin$,
- B is the size of the batch and Nin is the number of input features to the model.

```
# In[9]:
x = torch.ones(10, 1)
linear_model(x)

# Out[9]:
tensor([[0.6814],
        [0.6814],
        [0.6814],
        [0.6814],
        [0.6814],
        [0.6814],
        [0.6814],
        [0.6814],
        [0.6814],
        [0.6814]], grad_fn=<AddmmBackward>)
```

Why Batching

1- One big motivation is to make sure the computation we're asking for is big enough to saturate the computing resources we're using to perform the computation.

2- statistical information from the entire batch, and those statistics get better with larger batch sizes (input normalization)

We reshape our *B* inputs to *B* × *Nin*, where *Nin* is 1.

```
# In[2]:
t_c = [0.5,  14.0, 15.0, 28.0, 11.0,  8.0,  3.0, -4.0,  6.0, 13.0, 21.0]
t_u = [35.7, 55.9, 58.2, 81.9, 56.3, 48.9, 33.9, 21.8, 48.4, 60.4, 68.4]
t_c = torch.tensor(t_c).unsqueeze(1)
t_u = torch.tensor(t_u).unsqueeze(1)

t_u.shape

# Out[2]:
torch.Size([11, 1])
```

Adds the extra dimension at axis 1

Rebuilding our Linea Model with nn Module

```
# In[10]:
linear_model = nn.Linear(1, 1)
optimizer = optim.SGD(
    linear_model.parameters(),
    lr=1e-2)
```

This is just a redefinition from earlier.

This method call replaces [params].

```
# In[13]:
def training_loop(n_epochs, optimizer, model, loss_fn, t_u_train, t_u_val,
                  t_c_train, t_c_val):
    for epoch in range(1, n_epochs + 1):
        t_p_train = model(t_u_train)
        loss_train = loss_fn(t_p_train, t_c_train)

        t_p_val = model(t_u_val)

    loss_val = loss_fn(t_p_val, t_c_val)

    optimizer.zero_grad()
    loss_train.backward()
    optimizer.step()

    if epoch == 1 or epoch % 1000 == 0:
        print(f"Epoch {epoch}, Training loss {loss_train.item():.4f},"
              f" Validation loss {loss_val.item():.4f}")
```

The model is now passed in, instead of the individual params.

The loss function is also passed in. We'll use it in a moment.

It hasn't changed practically at all, except that now we don't pass `params` explicitly to `model` since the model itself holds its `Parameters` internally.

Rebuilding our Linea Model with nn Module

- `nn` comes with several common loss functions, among them `nn.MSELoss` (MSE stands for Mean Square Error), which is exactly what we defined earlier as our `loss_fn`.
- Loss functions in `nn` are still subclasses of `nn.Module`
- In our case, we get rid of the handwritten `loss_fn` and replace it:

```
# In[15]:
linear_model = nn.Linear(1, 1)
optimizer = optim.SGD(linear_model.parameters(), lr=1e-2)

training_loop(
    n_epochs = 3000,
    optimizer = optimizer,
    model = linear_model,
    loss_fn = nn.MSELoss(),
    t_u_train = t_un_train,
    t_u_val = t_un_val,
    t_c_train = t_c_train,
    t_c_val = t_c_val)

print()
print(linear_model.weight)
print(linear_model.bias)
```

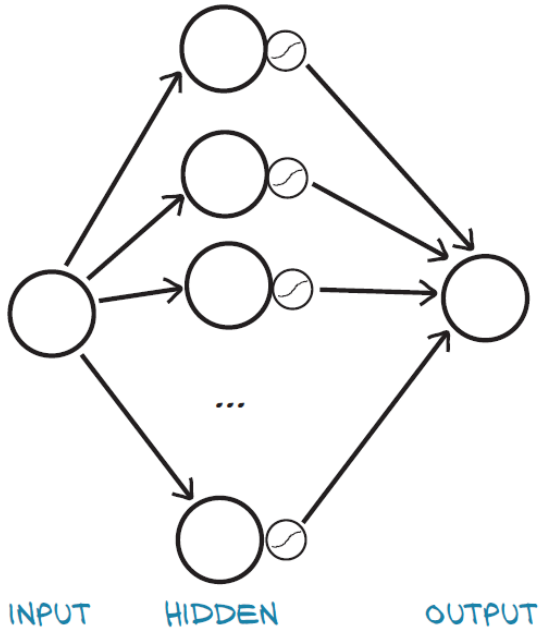
We are no longer using our hand-written loss function from earlier.

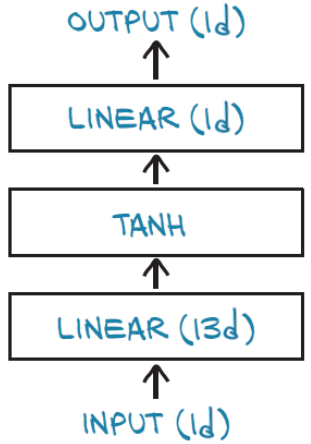
```
# Out[15]:
Epoch 1, Training loss 134.9599, Validation loss 183.1707
Epoch 1000, Training loss 4.8053, Validation loss 4.7307
Epoch 2000, Training loss 3.0285, Validation loss 3.0889
Epoch 3000, Training loss 2.8569, Validation loss 3.9105

Parameter containing:
tensor([[5.4319]], requires_grad=True)
Parameter containing:
tensor([-17.9693], requires_grad=True)
```

- Everything else input into our training loop stays the same. Even our results remain the same as before
- Getting the same results is expected, as a difference would imply a bug in one of the two implementations

Hidden Layer

- Let's build the simplest possible neural network: a linear module, followed by an activation function, feeding into another linear module.
- The **linear + activation layer** is commonly referred to as a ***hidden* layer**, since its outputs are not observed directly but fed into the output layer.
- While the input and output of the model are both of size 1, the size of the output of the first linear module (hidden layer) is usually larger than 1.
- This can lead different units to respond to different ranges of the input, which increases the capacity of our model.





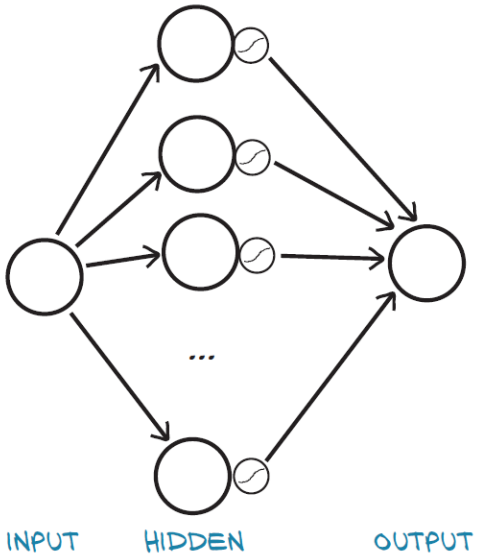
Our First Real Neural Network

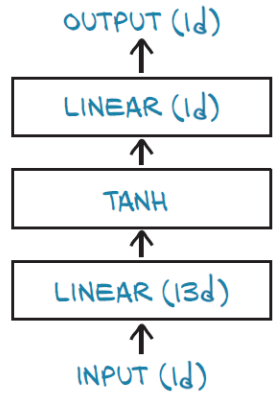
- `nn.Sequential` provides a simple way to concatenate modules.
- 1 input feature to 13 hidden features, passes them through a `tanh` activation, and linearly combines the resulting 13 numbers into 1 output feature.

```
# In[16]:
seq_model = nn.Sequential(
            nn.Linear(1, 13),
            nn.Tanh(),
            nn.Linear(13, 1))
seq_model

# Out[16]:
Sequential(
  (0): Linear(in_features=1, out_features=13, bias=True)
  (1): Tanh()
  (2): Linear(in_features=13, out_features=1, bias=True)
)
```

We chose 13 arbitrarily. We wanted a number that was a different size from the other tensor shapes we have floating around.

This 13 must match the first size, however.





- The end result is a model that takes the inputs expected by the first module specified as an argument of `nn.Sequential`, passes intermediate outputs to subsequent modules, and produces the output returned by the last module.

Inspecting the Parameters

- Calling `model.parameters()` will collect `weight` and `bias` from both the first and second linear modules

```
# In[17]:
[param.shape for param in seq_model.parameters()]

# Out[17]:
[torch.Size([13, 1]), torch.Size([13]), torch.Size([1, 13]), torch.Size([1])]
```

```
# In[18]:
for name, param in seq_model.named_parameters():
    print(name, param.shape)

# Out[18]:
0.weight torch.Size([13, 1])
0.bias torch.Size([13])
2.weight torch.Size([1, 13])
2.bias torch.Size([1])
```

- These are the tensors that the optimizer will get.
- After we call `model.backward()`, all parameters are populated with their `grad`, and the optimizer then updates their values accordingly during the `optimizer.step()` call.

Inspecting the Parameters

`Sequential` also accepts an `OrderedDict`, in which we can name each module passed to `Sequential`

```
# In[19]:
from collections import OrderedDict

seq_model = nn.Sequential(OrderedDict([
    ('hidden_linear', nn.Linear(1, 8)),
    ('hidden_activation', nn.Tanh()),
    ('output_linear', nn.Linear(8, 1))
]))

seq_model

# Out[19]:
Sequential(
  (hidden_linear): Linear(in_features=1, out_features=8, bias=True)
  (hidden_activation): Tanh()
  (output_linear): Linear(in_features=8, out_features=1, bias=True)
)
```

Inspecting the Parameters

- We can get more explanatory names for submodules:

```
# In[20]:
for name, param in seq_model.named_parameters():
    print(name, param.shape)

# Out[20]:
hidden_linear.weight torch.Size([8, 1])
hidden_linear.bias torch.Size([8])
output_linear.weight torch.Size([1, 8])
output_linear.bias torch.Size([1])
```

- We can also access a particular `Parameter` by using submodules as attributes

```
# In[21]:
seq_model.output_linear.bias

# Out[21]:
Parameter containing:
tensor([-0.0173], requires_grad=True)
```

- This is useful for inspecting parameters or their gradients: for instance, to monitor gradients during training, as we did at the beginning of this chapter.

Running our First Real Neural Network

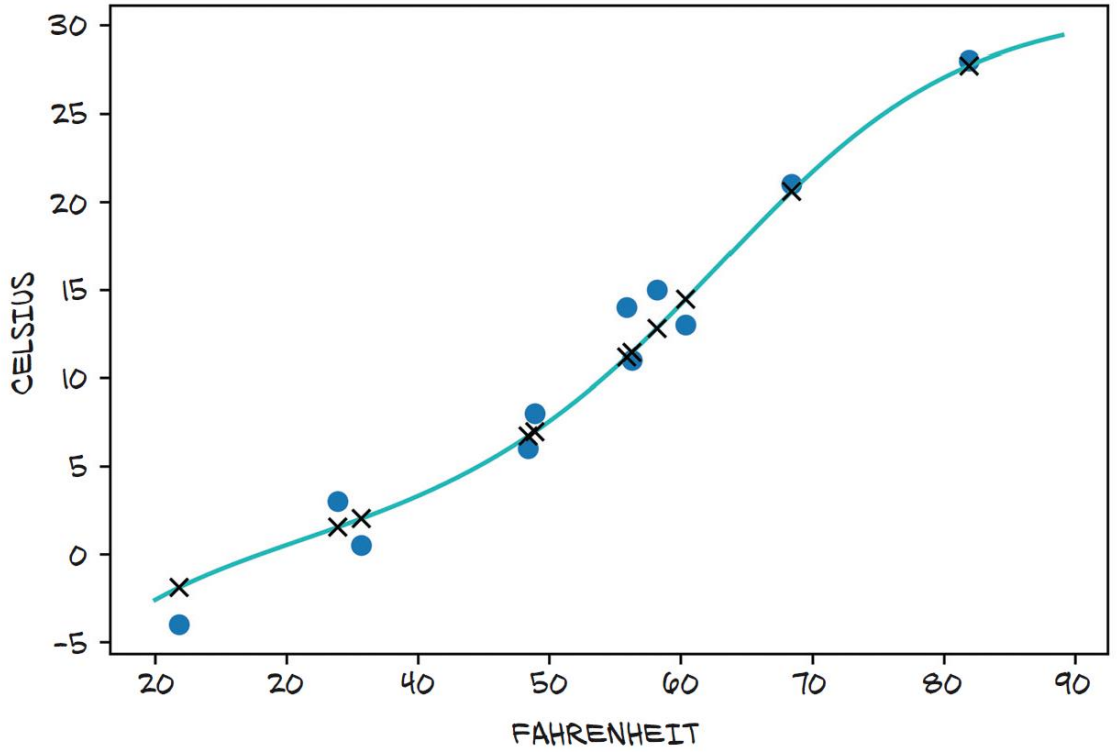
```
# In[22]:
optimizer = optim.SGD(seq_model.parameters(), lr=1e-3)

training_loop(
    n_epochs = 5000,
    optimizer = optimizer,
    model = seq_model,
    loss_fn = nn.MSELoss(),
    t_u_train = t_un_train,
    t_u_val = t_un_val,
    t_c_train = t_c_train,
    t_c_val = t_c_val)

print('output', seq_model(t_un_val))
print('answer', t_c_val)
print('hidden', seq_model.hidden_linear.weight.grad)
```

We've dropped the learning rate a bit to help with stability.

```
# Out[22]:
Epoch 1, Training loss 182.9724, Validation loss 231.8708
Epoch 1000, Training loss 6.6642, Validation loss 3.7330
Epoch 2000, Training loss 5.1502, Validation loss 0.1406
Epoch 3000, Training loss 2.9653, Validation loss 1.0005
Epoch 4000, Training loss 2.2839, Validation loss 1.6580
Epoch 5000, Training loss 2.1141, Validation loss 2.0215
output tensor([[-1.9930],
        [20.8729]], grad_fn=<AddmmBackward>)
answer tensor([[-4.],
        [21.]])
hidden tensor([[ 0.0272],
        [ 0.0139],
        [ 0.1692],
        [ 0.1735],
        [-0.1697],
        [ 0.1455],
        [-0.0136],
        [-0.0554]])
```

Plotting the Results

```
# In[23]:
from matplotlib import pyplot as plt

t_range = torch.arange(20., 90.).unsqueeze(1)

fig = plt.figure(dpi=600)
```




```
plt.xlabel("Fahrenheit")
plt.ylabel("Celsius")
plt.plot(t_u.numpy(), t_c.numpy(), 'o')
plt.plot(t_range.numpy(), seq_model(0.1 * t_range).detach().numpy(), 'c-')
plt.plot(t_u.numpy(), seq_model(0.1 * t_u).detach().numpy(), 'kx')
```

The WILLIAM STATES LEE COLLEGE *of* ENGINEERING
UNC CHARLOTTE

Image Processing Datasets: CIFAR-10

- CIFAR-10 consists of 60,000 tiny 32 × 32 color (RGB) images, labeled with an integer corresponding to 1 of 10 classes:
- airplane (0), automobile (1), bird (2), cat (3), deer (4), dog (5), frog (6), horse (7), ship (8), and truck (9).
- Nowadays, CIFAR-10 is considered too simple for developing or validating new research, but it serves our learning purposes just fine.

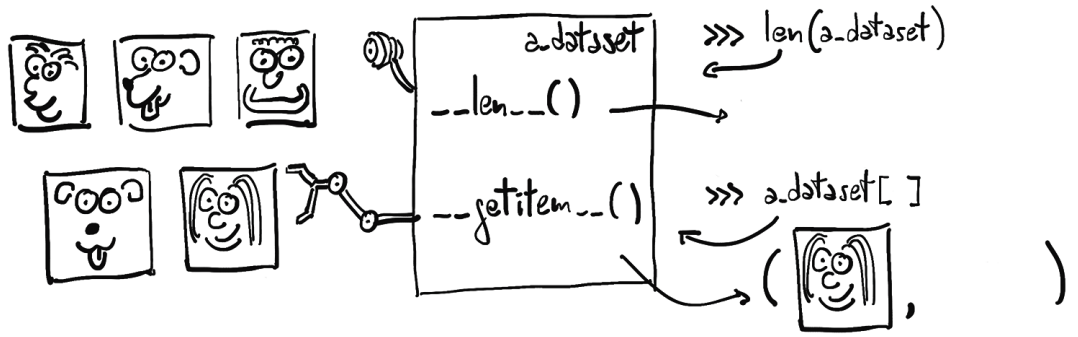
Downloading CIFAR-10

Instantiates a dataset for the training data; TorchVision downloads the data if it is not present.

With train=False, this gets us a dataset for the validation data, again downloading as necessary.

```
# In[2]:
from torchvision import datasets
data_path = '../data-unversioned/p1ch7/'
cifar10 = datasets.CIFAR10(data_path, train=True, download=True)
cifar10_val = datasets.CIFAR10(data_path, train=False, download=True)
```

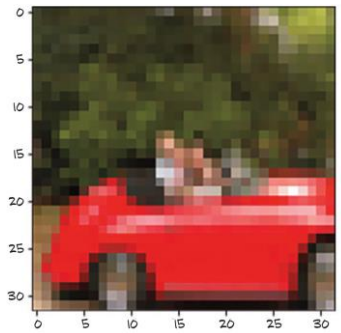
- The dataset is returned as a subclass of `torch.utils.data.Dataset`.
- It is an object that is required to implement two methods: `__len__` and `__getitem__`.
- `__len__` returns the number of items in the dataset
- `__getitem__` returns the item, consisting of a sample and its corresponding label (an integer index).



```
# In[5]:
len(cifar10)

# Out[5]:
50000
```

- We get a `PIL` (Python Imaging Library, the `PIL` package) image with our desired output—an integer with the value `1`, corresponding to “automobile”:

```
# In[6]:
img, label = cifar10[99]
img, label, class_names[label]

# Out[6]:
(<PIL.Image.Image image mode=RGB size=32x32 at 0x7FB383657390>,
 1,
 'automobile')

 # In[7]:
 plt.imshow(img)
 plt.show()
```



- Since the dataset is equipped with the `__getitem__` method, we can use the
- standard subscript for indexing tuples and lists to access individual items.

Dataset transforms

- We need a way to convert the PIL image to a PyTorch tensor before we can do anything with it.
- That's where `torchvision.transforms` comes in.
- This module defines a set of composable, function-like objects that can be passed as an argument to a `torchvision.`
- We can see the list of available objects as follows:

```
# In[8]:
from torchvision import transforms
dir(transforms)

# Out[8]:
['CenterCrop',
 'ColorJitter',
 ...
 'Normalize',
 'Pad',
 'RandomAffine',
 ...
 'RandomResizedCrop',
 'RandomRotation',
 'RandomSizedCrop',
 ...
 'TenCrop',
 'ToPILImage',
 'ToTensor',
 ...
]
```

ToTensor

- Among those transforms, we can spot `ToTensor`, which turns NumPy arrays and PIL images to tensors.
- It also takes care to lay out the dimensions of the output tensor as $C \times H \times W$ (channel, height, width)
- Once instantiated, it can be called like a function with the PIL image as the argument, returning a tensor as output:

```
# In[9]:
from torchvision import transforms

to_tensor = transforms.ToTensor()
img_t = to_tensor(img)
img_t.shape

# Out[9]:
torch.Size([3, 32, 32])
```

- The image has been turned into a 3 × 32 × 32 tensor and therefore a 3-channel (RGB) 32 × 32 image.
- Note that nothing has happened to `label`; it is still an integer.

ToTensor

- As we anticipated, we can pass the transform directly as an argument to `dataset.CIFAR10`:

```
# In[10]:
tensor_cifar10 = datasets.CIFAR10(data_path, train=True, download=False,
                                  transform=transforms.ToTensor())
```

- At this point, accessing an element of the dataset will return a tensor, rather than a PIL image:

```
# In[11]:
img_t, _ = tensor_cifar10[99]
type(img_t)

# Out[11]:
torch.Tensor

# In[12]:
img_t.shape, img_t.dtype

# Out[12]:
(torch.Size([3, 32, 32]), torch.float32)
```

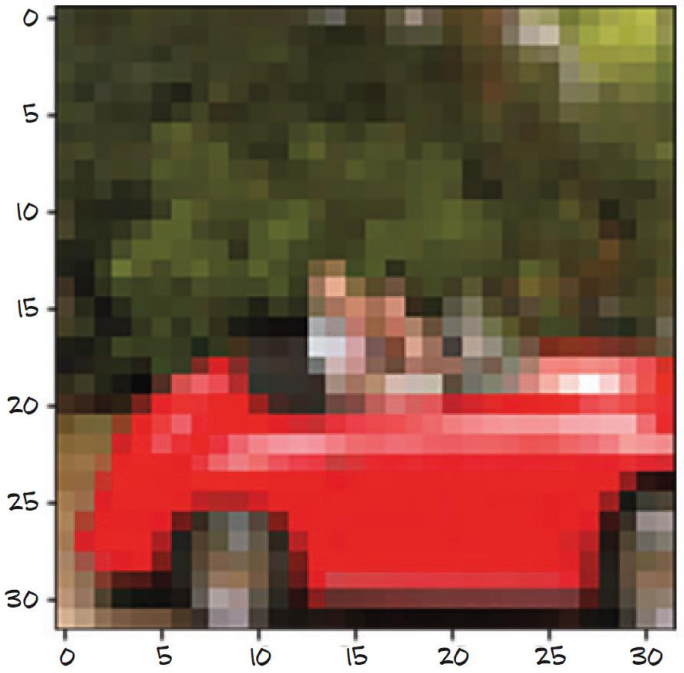
Whereas the values in the original PIL image ranged from 0 to 255 (8 bits per channel), the `ToTensor` transform turns the data into a 32-bit floating-point per channel, scaling the values down from 0.0 to 1.0.

ToTensor

- And let's verify that we're getting the same image out:

```
# In[14]:
plt.imshow(img_t.permute(1, 2, 0))
plt.show()

# Out[14]:
<Figure size 432x288 with 1 Axes>
```

Changes the order of the axes from C × H × W to H × W × C



- Note how we have to use `permute` to change the order of the axes from C × H × W to H × W × C to match what Matplotlib expects.

Normalizing Image Data

- It's good practice to normalize the dataset so that each channel has zero mean and unitary standard deviation.
- By choosing activation functions that are linear around 0 plus or minus 1 (or 2), keeping the data in the same range means it's more likely that neurons have nonzero gradients and, hence, will learn sooner.
- Also, normalizing each channel so that it has the same distribution will ensure that channel information can be mixed and updated through gradient descent using the same learning rate.
- In order to make it so that each channel has zero mean and unitary standard deviation, we can compute the mean value and the standard deviation of each channel across the dataset and apply the following transform: `v_n[c] = (v[c] - mean[c]) / stdev[c].`

Normalizing Image Data

- Since the CIFAR-10 dataset is small, we'll be able to manipulate it entirely in memory.
- Let's stack all the tensors returned by the dataset along an extra dimension:

```
# In[15]:
imgs = torch.stack([img_t for img_t, _ in tensor_cifar10], dim=3)
imgs.shape

# Out[15]:
torch.Size([3, 32, 32, 50000])
```

```
# In[16]:
imgs.view(3, -1).mean(dim=1)

# Out[16]:
tensor([0.4915, 0.4823, 0.4468])
```

Recall that view(3, -l) keeps the three channels and merges all the remaining dimensions into one, figuring out the appropriate size. Here our 3 × 32 × 32 image is transformed into a 3 × 1,024 vector, and then the mean is taken over the 1,024 elements of each channel.

```
# In[17]:
imgs.view(3, -1).std(dim=1)

# Out[17]:
tensor([0.2470, 0.2435, 0.2616])
```

Normalizing Image Data

- With these numbers in our hands, we can initialize the `Normalize` transform

```
# In[18]:
transforms.Normalize((0.4915, 0.4823, 0.4468), (0.2470, 0.2435, 0.2616))

# Out[18]:
Normalize(mean=(0.4915, 0.4823, 0.4468), std=(0.247, 0.2435, 0.2616))
```

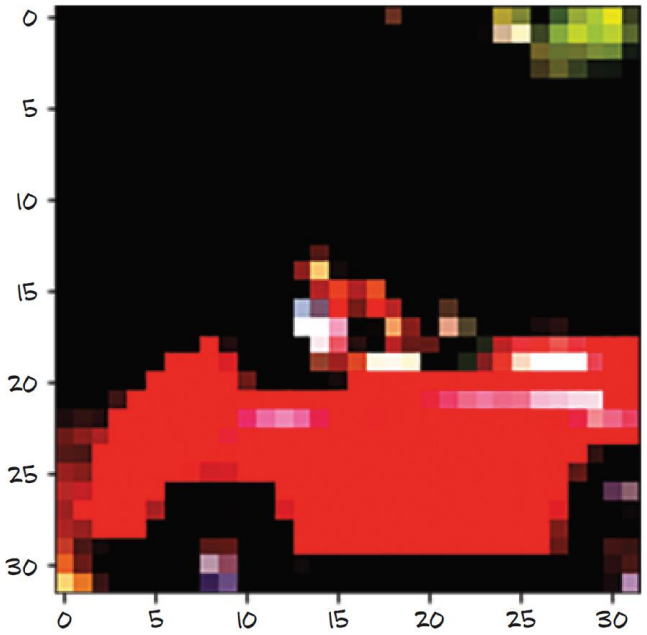
- and concatenate it after the `ToTensor` transform:

```
# In[19]:
transformed_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))
]))
```

Normalizing Image Data

- Note that, at this point, plotting an image drawn from the dataset won't provide us with a faithful representation of the actual image:

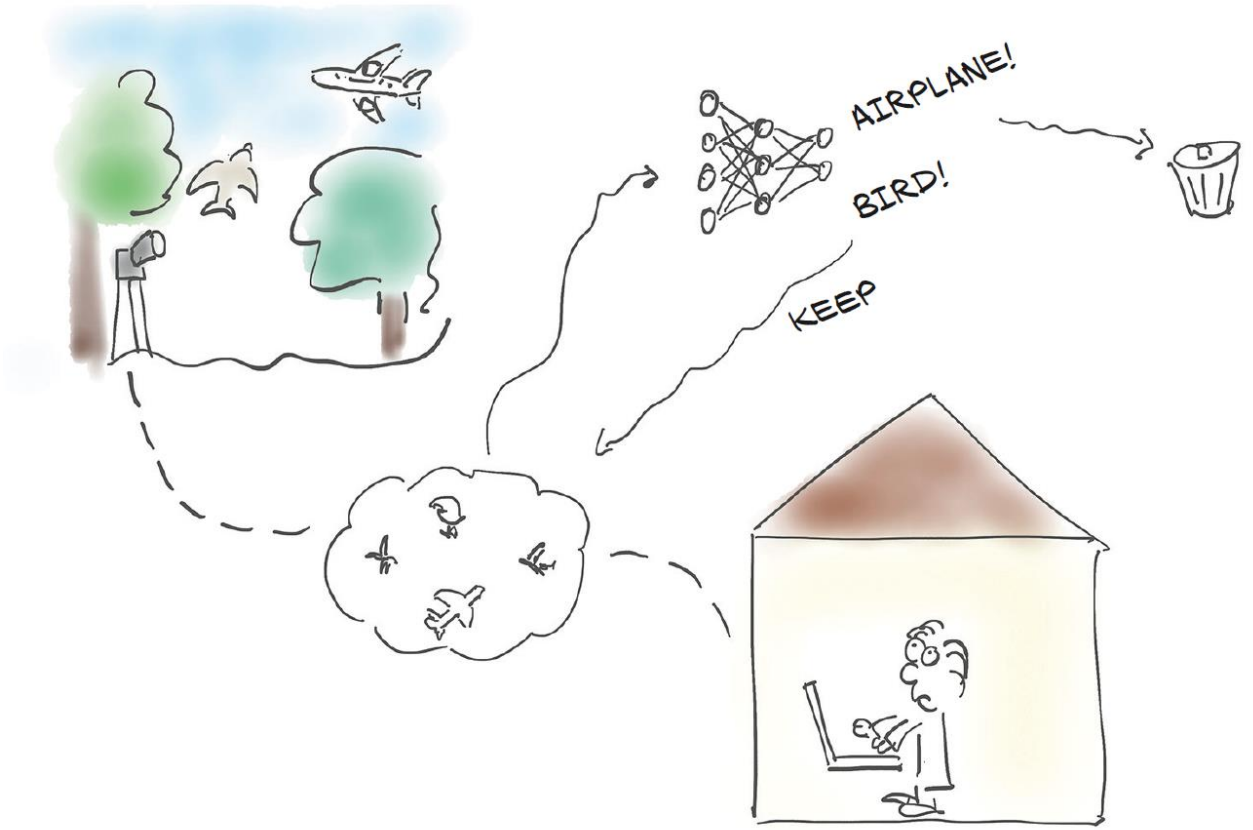
```
# In[21]:
img_t, _ = transformed_cifar10[99]

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



- Normalization has shifted the RGB levels outside the 0.0 to 1.0 range and changed the overall magnitudes of the channels.
- All of the data is still there; it's just that Matplotlib renders it as black.

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.



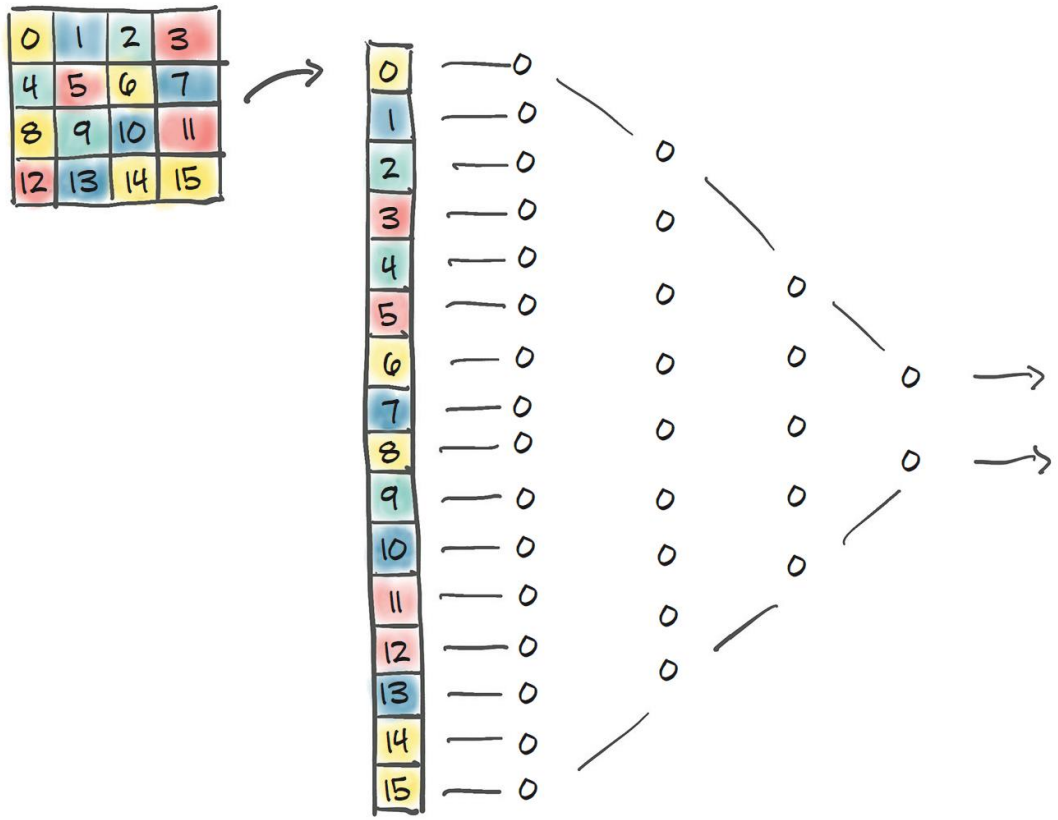
Distinguishing birds from airplanes: Building the Dataset

- 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,
                512,
            ),
            nn.Tanh(),
            nn.Linear(
                512,
                n_out,
            )
        )
```

Input features → 3072

512 → Hidden layer size

Output classes → n_out

- 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

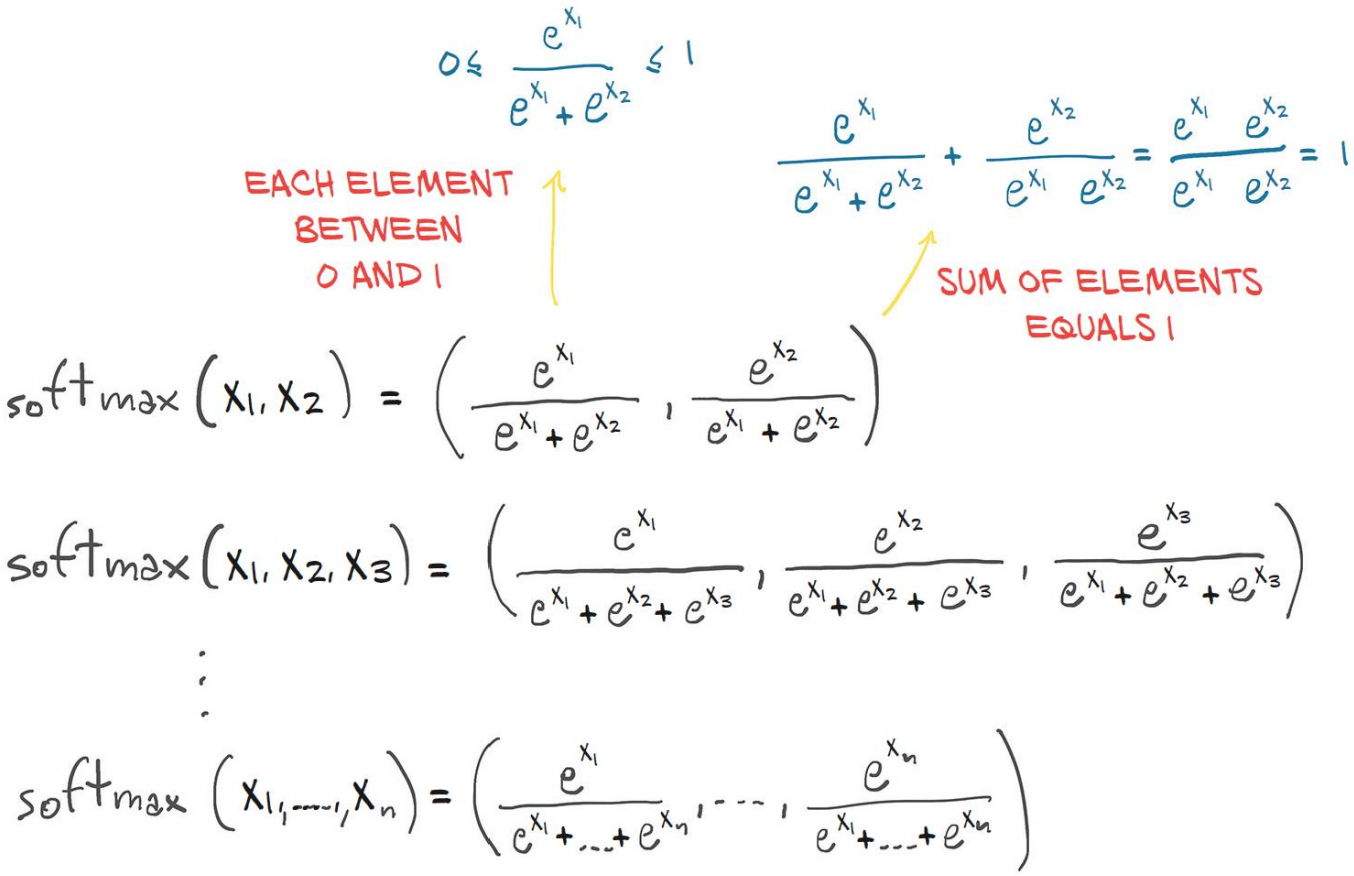
Distinguishing birds from airplanes: Output of the Classifier

- **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.



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))
```

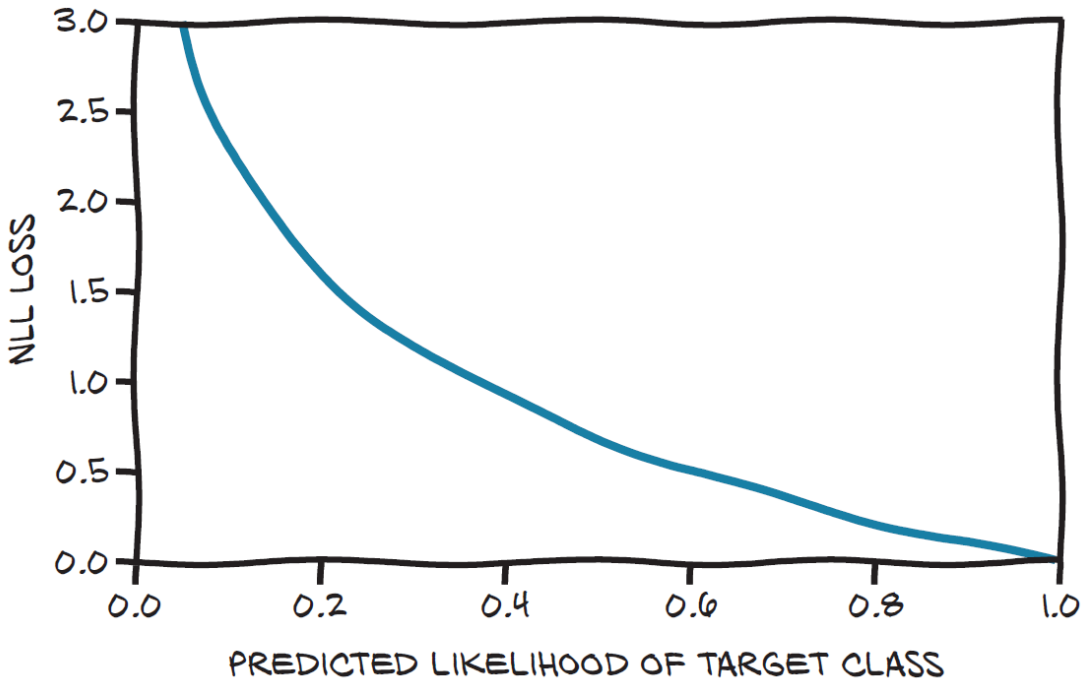
A loss for classifying

- **MSE:** We could still use MSE and make our output probabilities converge to `[0.0, 1.0]` and `[1.0, 0.0]`.

- **Alternative: cross-entropy loss:** a loss function that is very high when the likelihood is low, so low that the alternatives have a higher probability. Conversely, the loss should be low when the likelihood is higher than the alternatives.

Negative log likelihood (NLL)

- NLL = - sum(log(out_i[c_i]))
- where the sum is taken over 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)))
```

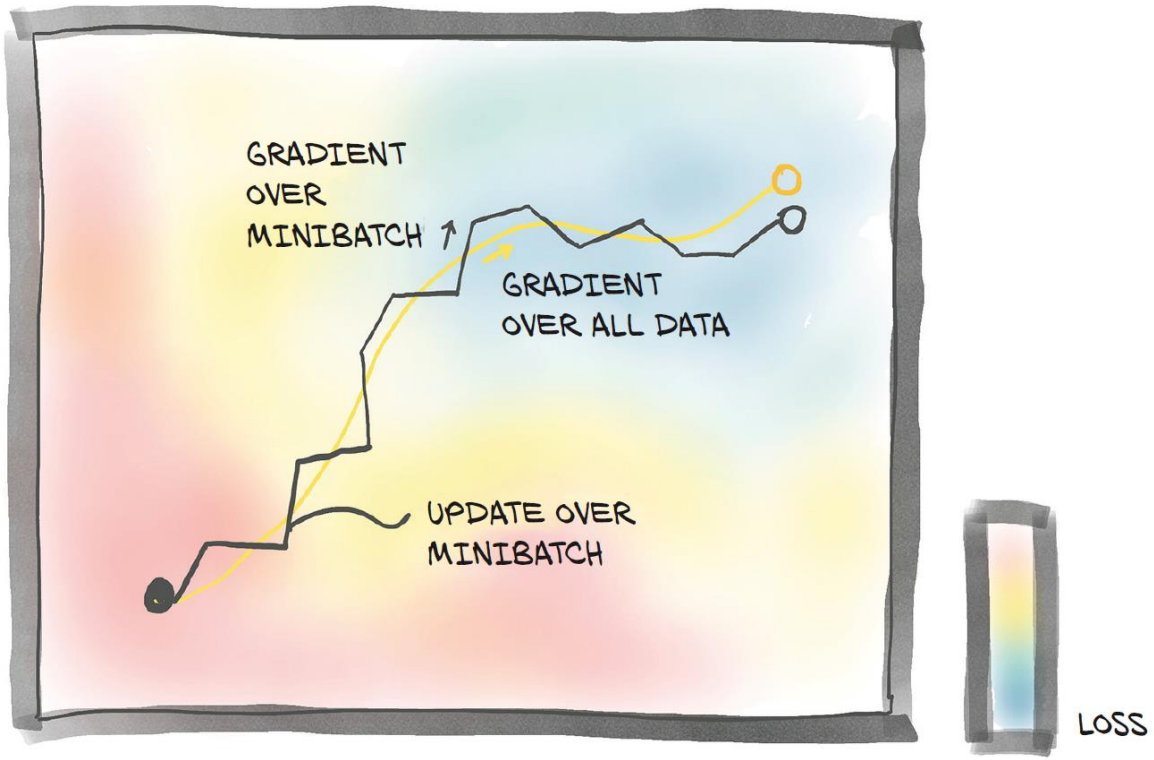
Prints the loss for the last image. In the next chapter, we will improve our output to give an average over the entire epoch.

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.

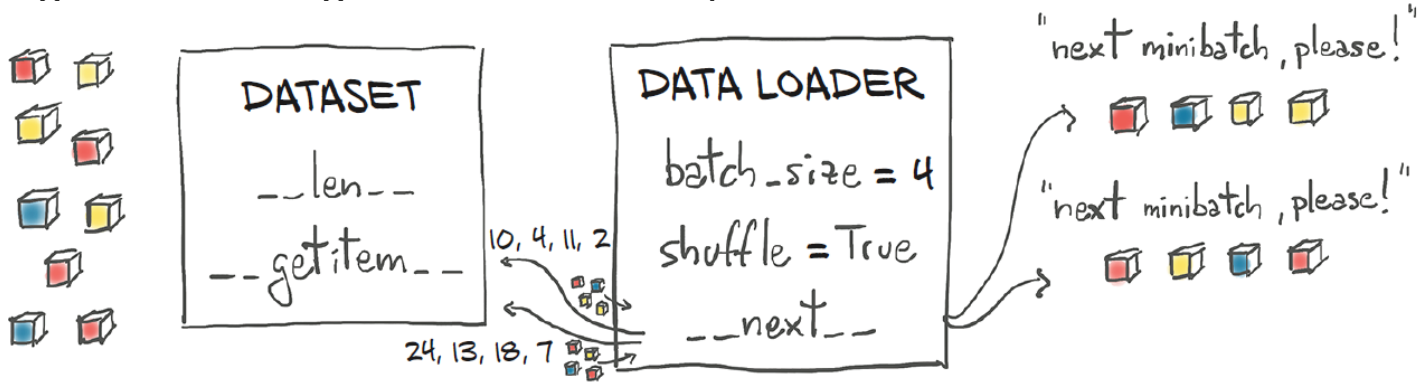
Averaging updates over minibatches

- 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.

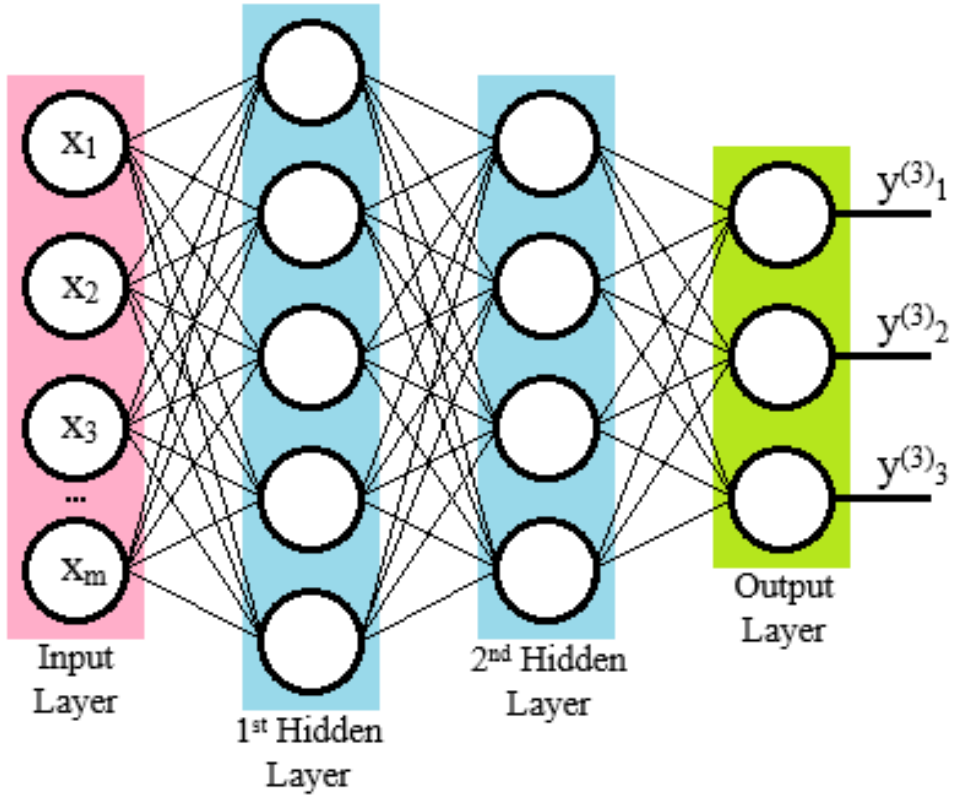
- We can compute the accuracy of our model on the validation, set in terms of the number of correct classifications over the total:

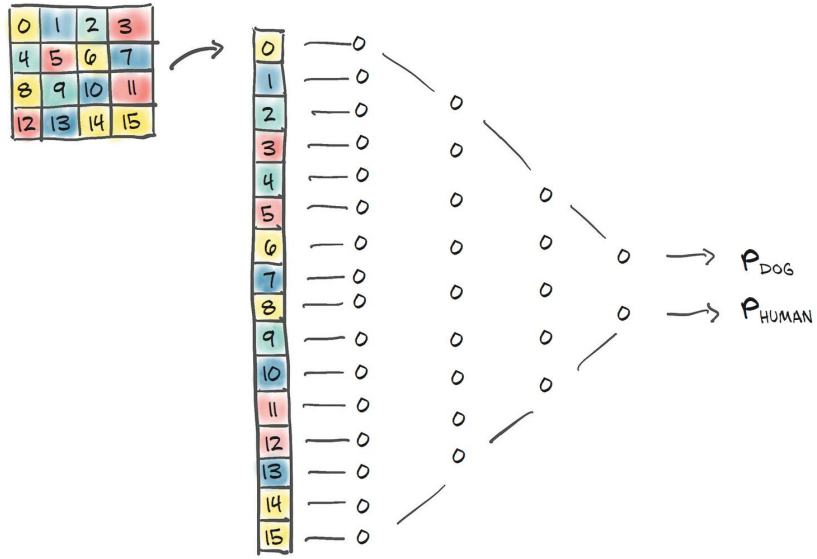
```
val_loader = torch.utils.data.DataLoader(cifar2_val, batch_size=64,
                                         shuffle=False)

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
```

The WILLIAM STATES LEE COLLEGE of ENGINEERING
UNC CHARLOTTE

Fully Connected Neural Networks





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

The WILLIAM STATES LEE COLLEGE of ENGINEERING
UNC CHARLOTTE

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])
```

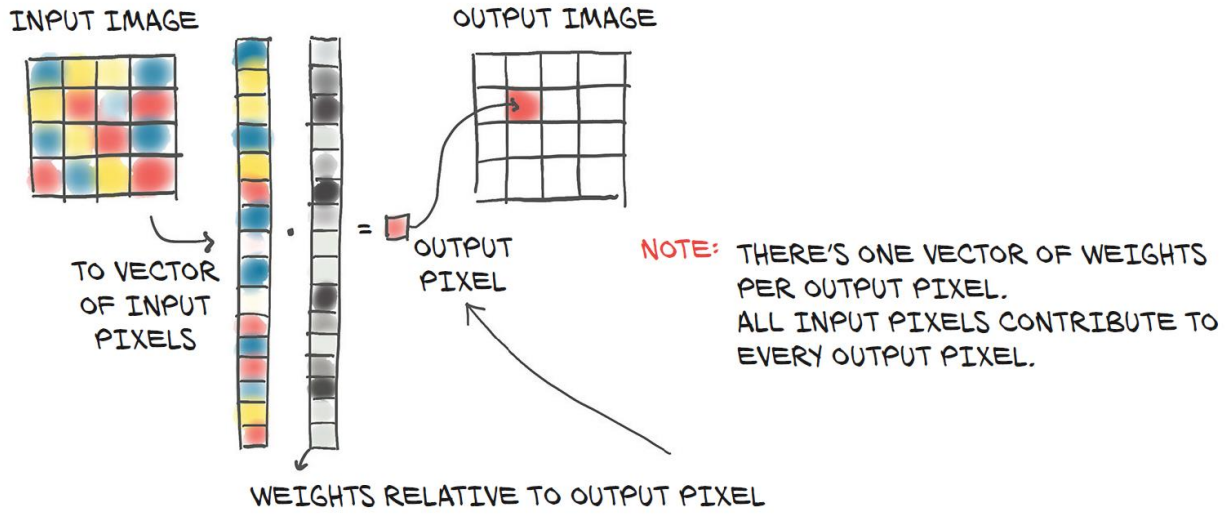
Even our first network was pretty large, with 1.5 M parameters

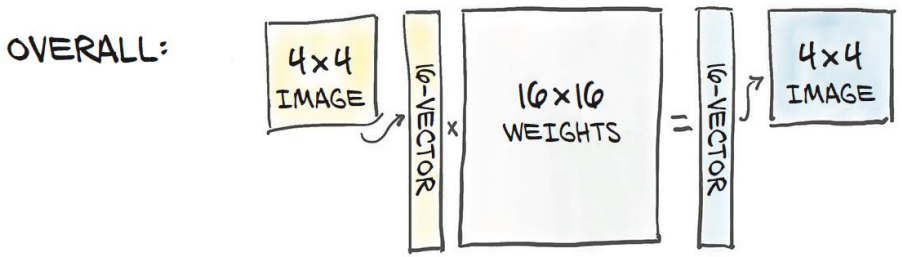
```
# In[10]:
linear = nn.Linear(3072, 1024)

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.

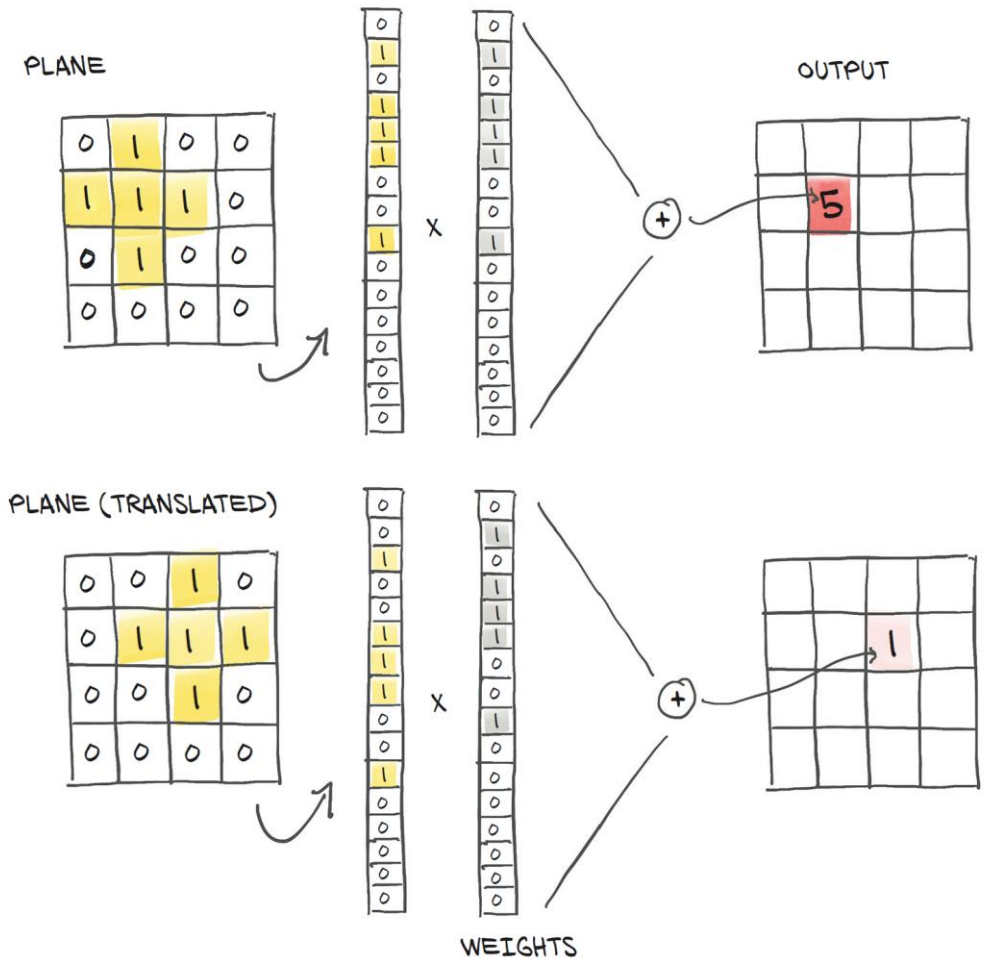
The WILLIAM STATES LEE COLLEGE of ENGINEERING
UNC CHARLOTTE

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*.