

# Training Deep Neural Networks on a GPU with PyTorch

## Part 4 of "PyTorch: Zero to GANs"

This notebook is the fourth in a series of tutorials on building deep learning models with PyTorch, an open source neural networks library. Check out the full series:

1. [PyTorch Basics: Tensors & Gradients](#)
2. [Linear Regression & Gradient Descent](#)
3. [Image Classification using Logistic Regression](#)
4. [Training Deep Neural Networks on a GPU](#)
5. [Image Classification using Convolutional Neural Networks](#)
6. [Data Augmentation, Regularization and ResNets](#)
7. [Generating Images using Generative Adversarial Networks](#)

In [the previous tutorial](#), we trained a logistic regression model to identify handwritten digits from the MNIST dataset with an accuracy of around 86%.



However, we also noticed that it's quite difficult to improve the accuracy beyond 87%, due to the limited power of the model. In this post, we'll try to improve upon it using a *feedforward neural network*.

## System Setup

This tutorial takes a code-first approach, and you should try to follow along by running and experimenting with the code yourself. The easiest way to start executing this notebook is to click the **"Run"** button at the top of this page, and select **"Run on Kaggle"**. This will run the notebook on Kaggle, a free online service for running Jupyter notebooks (you might need to create an account).

## Running on your computer locally

(Skip this if you are running on Kaggle) You can clone this notebook, install the required dependencies using conda, and start Jupyter by running the following commands on the terminal:

```
pip install jovian --upgrade           # Install the jovian library
jovian clone aakashns/04-feedforward-nn # Download notebook
cd 04-feedforward-nn                  # Enter the created directory
conda create -n 04-feedforward-nn python=3.8 # Create a conda environment
conda activate 04-feedforward-nn          # Activate virtual environment
conda install jupyter                  # Install Jupyter
jupyter notebook                       # Start Jupyter
```

On older versions of conda, you might need to run `source activate 04-feedforward-nn` to activate the virtual environment. For a more detailed explanation of the above steps, check out the System setup section in [the first notebook](#).

## Preparing the Data

## Preparing the Data

The data preparation is identical to the [previous tutorial](#). We begin by importing the required modules & classes.

In [ ]:

```
# Uncomment and run the commands below if imports fail
# !conda install numpy pandas pytorch torchvision cpuonly -c pytorch -y
# !pip install matplotlib --upgrade --quiet
```

In [1]:

```
import torch
import torchvision
import numpy as np
import matplotlib.pyplot as plt
import torch.nn as nn
import torch.nn.functional as F
from torchvision.datasets import MNIST
from torchvision.transforms import ToTensor
from torchvision.utils import make_grid
from torch.utils.data.dataloader import DataLoader
from torch.utils.data import random_split
%matplotlib inline
```

We download the data and create a PyTorch dataset using the `MNIST` class from `torchvision.datasets`.

In [2]:

```
dataset = MNIST(root='data/', download=True, transform=ToTensor())
```

Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to data/MNIST/raw/train-images-idx3-ubyte.gz

Extracting data/MNIST/raw/train-images-idx3-ubyte.gz to data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to data/MNIST/raw/train-labels-idx1-ubyte.gz

Extracting data/MNIST/raw/train-labels-idx1-ubyte.gz to data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to data/MNIST/raw/t10k-images-idx3-ubyte.gz

Extracting data/MNIST/raw/t10k-images-idx3-ubyte.gz to data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to data/MNIST/raw/t10k-labels-idx1-ubyte.gz

Extracting data/MNIST/raw/t10k-labels-idx1-ubyte.gz to data/MNIST/raw

Processing...

Done!

/opt/conda/conda-bld/pytorch\_1587428398394/work/torch/csrc/utils/tensor\_numpy.cpp:141: UserWarning: The given NumPy array is not writeable, and PyTorch does not support non-writeable tensors. This means you can write to the underlying (supposedly non-writeable) NumPy array using the tensor. You may want to copy the array to protect its data or make it writeable before converting it to a tensor. This type of warning will be suppressed for the rest of this program.

Next, let's use the `random_split` helper function to set aside 10000 images for our validation set.

In [3]:

```
val_size = 10000
train_size = len(dataset) - val_size

train_ds, val_ds = random_split(dataset, [train_size, val_size])
len(train_ds), len(val_ds)
```

```
len(train_ds), len(val_ds)
```

```
Out[3]:  
(50000, 10000)
```

We can now create PyTorch data loaders for training and validation.

```
In [4]:  
batch_size=128
```

```
In [5]:  
train_loader = DataLoader(train_ds, batch_size, shuffle=True, num_workers=4, pin_memory=True)  
val_loader = DataLoader(val_ds, batch_size*2, num_workers=4, pin_memory=True)
```

Can you figure out what the `num_workers` and `pin_memory` are used for? Try looking into the documentation:  
<https://pytorch.org/docs/stable/data.html>

Let's visualize a batch of data in a grid using the `make_grid` function from `torchvision`. We'll also use the `.permute` method on the tensor to move the channels to the last dimension, as expected by `matplotlib`.

```
In [6]:  
for images, _ in train_loader:  
    print('images.shape:', images.shape)  
    plt.figure(figsize=(16,8))  
    plt.axis('off')  
    plt.imshow(make_grid(images, nrow=16).permute((1, 2, 0)))  
    break
```

```
images.shape: torch.Size([128, 1, 28, 28])
```



## Model

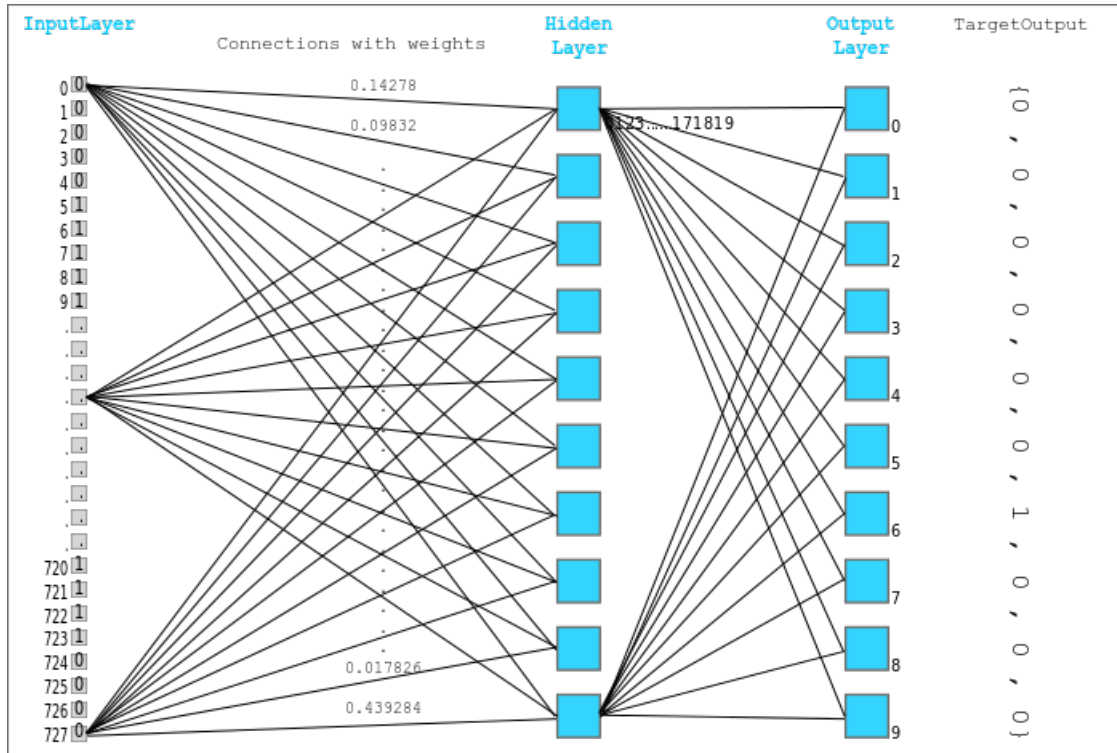
To improve upon [logistic regression](#), we'll create a neural network with one **hidden layer**. Here's what this means:

- Instead of using a single `nn.Linear` object to transform a batch of inputs (pixel intensities) into a batch of outputs (class probabilities), we'll use two `nn.Linear` objects. Each of these is called a layer in the network.
- The first layer (also known as the hidden layer) will transform the input matrix of shape `batch_size x 784` into an intermediate output matrix of shape `batch_size x hidden_size`, where `hidden_size` is a preconfigured parameter (e.g.

intermediate output matrix of shape `batch_size x hidden_size`, where `hidden_size` is a preconfigured parameter (e.g. 32 or 64).

- The intermediate outputs are then passed into a non-linear *activation function*, which operates on individual elements of the output matrix.
- The result of the activation function, which is also of size `batch_size x hidden_size`, is passed into the second layer (also known as the output layer), which transforms it into a matrix of size `batch_size x 10`, identical to the output of the logistic regression model.

Introducing a hidden layer and an activation function allows the model to learn more complex, multi-layered and non-linear relationships between the inputs and the targets. Here's what it looks like visually:



The activation function we'll use here is called a **Rectified Linear Unit** or **ReLU**, and it has a really simple formula:  $\text{relu}(x) = \max(0, x)$  i.e. if an element is negative, we replace it by 0, otherwise we leave it unchanged.

To define the model, we extend the `nn.Module` class, just as we did with logistic regression.

In [7]:

```
def accuracy(outputs, labels):
    _, preds = torch.max(outputs, dim=1)
    return torch.tensor(torch.sum(preds == labels).item() / len(preds))
```

In [8]:

```
class MnistModel(nn.Module):
    """Feedforward neural network with 1 hidden layer"""
    def __init__(self, in_size, hidden_size, out_size):
        super().__init__()
        # hidden layer
        self.linear1 = nn.Linear(in_size, hidden_size)
        # output layer
        self.linear2 = nn.Linear(hidden_size, out_size)

    def forward(self, xb):
        # Flatten the image tensors
        xb = xb.view(xb.size(0), -1)
        # Get intermediate outputs using hidden layer
        out = self.linear1(xb)
        # Apply activation function
        out = F.relu(out)
        # Get predictions using output layer
        out = self.linear2(out)
        return out

    def training_step(self, batch):
        images, labels = batch
```

```

images, labels = batch
out = self(images) # Generate predictions
loss = F.cross_entropy(out, labels) # Calculate loss
return loss

def validation_step(self, batch):
    images, labels = batch
    out = self(images) # Generate predictions
    loss = F.cross_entropy(out, labels) # Calculate loss
    acc = accuracy(out, labels) # Calculate accuracy
    return {'val_loss': loss, 'val_acc': acc}

def validation_epoch_end(self, outputs):
    batch_losses = [x['val_loss'] for x in outputs]
    epoch_loss = torch.stack(batch_losses).mean() # Combine losses
    batch_accs = [x['val_acc'] for x in outputs]
    epoch_acc = torch.stack(batch_accs).mean() # Combine accuracies
    return {'val_loss': epoch_loss.item(), 'val_acc': epoch_acc.item()}

def epoch_end(self, epoch, result):
    print("Epoch [{}], val_loss: {:.4f}, val_acc: {:.4f}".format(epoch, result['val_loss'], result[
'val_acc']))

```

We'll create a model that contains a hidden layer with 32 activations.

In [9]:

```

input_size = 784
hidden_size = 32 # you can change this
num_classes = 10

```

In [10]:

```

model = MnistModel(input_size, hidden_size=32, out_size=num_classes)

```

Let's take a look at the model's parameters. We expect to see one weight and bias matrix for each of the layers.

In [11]:

```

for t in model.parameters():
    print(t.shape)

```

```

torch.Size([32, 784])
torch.Size([32])
torch.Size([10, 32])
torch.Size([10])

```

Let's try and generate some outputs using our model. We'll take the first batch of 128 images from our dataset, and pass them into our model.

In [12]:

```

for images, labels in train_loader:
    outputs = model(images)
    loss = F.cross_entropy(outputs, labels)
    print('Loss:', loss.item())
    break

print('outputs.shape : ', outputs.shape)
print('Sample outputs :\n', outputs[:2].data)

```

```

Loss: 2.327113389968872
outputs.shape : torch.Size([128, 10])
Sample outputs :
tensor([[ -0.2116,  0.1456, -0.1273,  0.0666,  0.1468,  0.0490, -0.0862,
          -0.0270,  0.1383],
        [-0.1183, -0.0705,  0.1517, -0.2360,  0.0831,  0.0859,  0.0030, -0.0225,
          0.1167,  0.0651]])

```

## Using a GPU

As the sizes of our models and datasets increase, we need to use GPUs to train our models within a reasonable amount of time. GPUs contain hundreds of cores that are optimized for performing expensive matrix operations on floating point numbers in a short time, which makes them ideal for training deep neural networks with many layers. You can use GPUs for free on [Kaggle kernels](#) or [Google Colab](#), or rent GPU-powered machines on services like [Google Cloud Platform](#), [Amazon Web Services](#) or [Paperspace](#).

We can check if a GPU is available and the required NVIDIA CUDA drivers are installed using `torch.cuda.is_available`.

In [13]:

```
torch.cuda.is_available()
```

Out[13]:

True

Let's define a helper function to ensure that our code uses the GPU if available, and defaults to using the CPU if it isn't.

In [14]:

```
def get_default_device():
    """Pick GPU if available, else CPU"""
    if torch.cuda.is_available():
        return torch.device('cuda')
    else:
        return torch.device('cpu')
```

In [15]:

```
device = get_default_device()
device
```

Out[15]:

device(type='cuda')

Next, let's define a function that can move data and model to a chosen device.

In [16]:

```
def to_device(data, device):
    """Move tensor(s) to chosen device"""
    if isinstance(data, (list, tuple)):
        return [to_device(x, device) for x in data]
    return data.to(device, non_blocking=True)
```

In [17]:

```
for images, labels in train_loader:
    print(images.shape)
    images = to_device(images, device)
    print(images.device)
    break
```

```
torch.Size([128, 1, 28, 28])
cuda:0
```

Finally, we define a `DeviceDataLoader` class to wrap our existing data loaders and move data to the selected device, as a batches are accessed. Interestingly, we don't need to extend an existing class to create a PyTorch dataloader. All we need is an `__iter__` method to retrieve batches of data, and an `__len__` method to get the number of batches.

In [18]:

```

class DeviceDataLoader():
    """Wrap a dataloader to move data to a device"""
    def __init__(self, dl, device):
        self.dl = dl
        self.device = device

    def __iter__(self):
        """Yield a batch of data after moving it to device"""
        for b in self.dl:
            yield to_device(b, self.device)

    def __len__(self):
        """Number of batches"""
        return len(self.dl)

```

We can now wrap our data loaders using `DeviceDataLoader`.

In [19]:

```

train_loader = DeviceDataLoader(train_loader, device)
val_loader = DeviceDataLoader(val_loader, device)

```

Tensors that have been moved to the GPU's RAM have a `device` property which includes the word `cuda`. Let's verify this by looking at a batch of data from `valid_dl`.

In [20]:

```

for xb, yb in val_loader:
    print('xb.device:', xb.device)
    print('yb:', yb)
    break

```

```

xb.device: cuda:0
yb: tensor([6, 9, 7, 9, 3, 2, 5, 6, 9, 9, 3, 8, 2, 7, 1, 3, 9, 2, 6, 0, 6, 4, 1, 5,
           8, 3, 5, 8, 6, 1, 2, 3, 4, 1, 4, 2, 4, 6, 8, 1, 5, 9, 9, 5, 3, 7, 7, 3,
           3, 4, 4, 6, 2, 5, 1, 0, 9, 0, 0, 0, 8, 7, 3, 9, 9, 8, 1, 6, 1, 1, 7, 6,
           0, 6, 7, 8, 3, 2, 2, 3, 6, 7, 5, 7, 1, 4, 3, 3, 8, 6, 0, 7, 1, 1, 4, 1,
           2, 6, 0, 4, 8, 4, 0, 1, 7, 0, 0, 0, 0, 8, 9, 2, 9, 8, 7, 4, 0, 1, 8, 3,
           0, 8, 2, 4, 9, 2, 7, 7, 8, 8, 5, 6, 4, 5, 4, 9, 6, 7, 4, 3, 6, 1, 4, 0,
           7, 1, 4, 0, 4, 5, 3, 8, 1, 0, 2, 9, 1, 4, 1, 0, 0, 5, 1, 9, 4, 5, 8, 3,
           8, 9, 0, 8, 5, 4, 7, 8, 6, 0, 9, 7, 6, 6, 5, 1, 3, 4, 9, 6, 4, 5, 7, 3,
           0, 0, 2, 8, 6, 6, 4, 5, 5, 2, 5, 9, 0, 1, 9, 7, 2, 8, 5, 1, 5, 3, 2, 2,
           9, 6, 6, 8, 3, 2, 0, 5, 4, 7, 4, 6, 1, 0, 9, 8, 2, 2, 4, 1, 7, 6, 4, 6,
           2, 1, 7, 2, 9, 3, 0, 8, 6, 5, 4, 6, 4, 5, 8, 2], device='cuda:0')

```

## Training the Model

We can use the exact same training loops from the logistic regression notebook.

In [21]:

```

def evaluate(model, val_loader):
    outputs = [model.validation_step(batch) for batch in val_loader]
    return model.validation_epoch_end(outputs)

def fit(epochs, lr, model, train_loader, val_loader, opt_func=torch.optim.SGD):
    history = []
    optimizer = opt_func(model.parameters(), lr)
    for epoch in range(epochs):
        # Training Phase
        for batch in train_loader:
            loss = model.training_step(batch)
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()
        # Validation phase
        result = evaluate(model, val_loader)
        model.epoch_end(epoch, result)

```

```
        history.append(result)
    return history
```

We also define an `accuracy` function which calculates the overall accuracy of the model on an entire batch of outputs, so that we can use it as a metric in `fit`.

Before we train the model, we need to ensure that the data and the model's parameters (weights and biases) are on the same device (CPU or GPU). We can reuse the `to_device` function to move the model's parameters to the right device.

In [22]:

```
# Model (on GPU)
model = MnistModel(input_size, hidden_size=hidden_size, out_size=num_classes)
to_device(model, device)
```

Out[22]:

```
MnistModel(
  (linear1): Linear(in_features=784, out_features=32, bias=True)
  (linear2): Linear(in_features=32, out_features=10, bias=True)
)
```

Let's see how the model performs on the validation set with the initial set of weights and biases.

In [23]:

```
history = [evaluate(model, val_loader)]
history
```

Out[23]:

```
[{'val_loss': 2.3214690685272217, 'val_acc': 0.0771484375}]
```

The initial accuracy is around 10%, which is what one might expect from a randomly initialized model (since it has a 1 in 10 chance of getting a label right by guessing randomly).

We are now ready to train the model. Let's train for 5 epochs and look at the results. We can use a relatively higher learning of 0.5.

In [24]:

```
history += fit(5, 0.5, model, train_loader, val_loader)
```

```
Epoch [0], val_loss: 0.2516, val_acc: 0.9271
Epoch [1], val_loss: 0.2142, val_acc: 0.9334
Epoch [2], val_loss: 0.1676, val_acc: 0.9489
Epoch [3], val_loss: 0.1863, val_acc: 0.9429
Epoch [4], val_loss: 0.2458, val_acc: 0.9196
```

96% is pretty good! Let's train the model for 5 more epochs at a lower learning rate of 0.1, to further improve the accuracy.

In [25]:

```
history += fit(5, 0.1, model, train_loader, val_loader)
```

```
Epoch [0], val_loss: 0.1201, val_acc: 0.9667
Epoch [1], val_loss: 0.1175, val_acc: 0.9657
Epoch [2], val_loss: 0.1174, val_acc: 0.9659
Epoch [3], val_loss: 0.1160, val_acc: 0.9658
Epoch [4], val_loss: 0.1146, val_acc: 0.9665
```

We can now plot the losses & accuracies to study how the model improves over time.

In [26]:

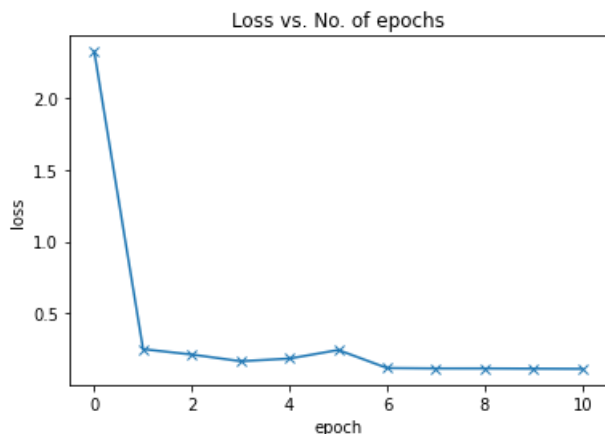
```
plt.plot(history['val_loss'], history['val_acc'])
```



```
import matplotlib.pyplot as plt
%matplotlib inline
```

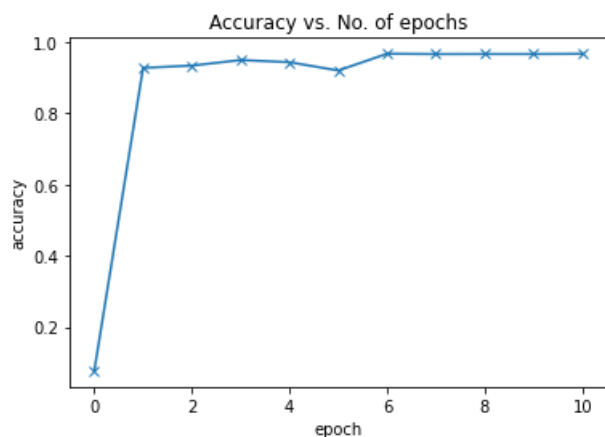
In [27]:

```
losses = [x['val_loss'] for x in history]
plt.plot(losses, '-x')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.title('Loss vs. No. of epochs');
```



In [28]:

```
accuracies = [x['val_acc'] for x in history]
plt.plot(accuracies, '-x')
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.title('Accuracy vs. No. of epochs');
```



Our current model outperforms the logistic regression model (which could only reach around 86% accuracy) by a huge margin! It quickly reaches an accuracy of 97%, but doesn't improve much beyond this. To improve the accuracy further, we need to make the model more powerful. As you can probably guess, this can be achieved by increasing the size of the hidden layer, or adding more hidden layers. I encourage you to try out both these approaches and see which one works better.

## Commit and upload the notebook

As a final step, we can save and commit our work using the jovian library.

In [29]:

```
!pip install jovian --upgrade -q
```

WARNING: You are using pip version 20.1; however, version 20.1.1 is available.

You should consider upgrading via the '!/opt/conda/bin/python3.7 -m pip install --upgrade pip' command

you should consider upgrading via the `!pip install --upgrade pip` command.

In [30]:

```
import jovian
```

In [ ]:

```
jovian.commit(project='04-feedforward-nn', environment=None)
```

[jovian] Attempting to save notebook..

## Summary and Further Reading

Here is a summary of the topics covered in this tutorial:

- We created a neural network with one hidden layer to improve upon the logistic regression model from the previous tutorial. We also used the ReLU activation function to introduce non-linearity into the model, allowing it to learn more complex relationships between the inputs (pixel densities) and outputs (class probabilities).
- We defined some utilities like `get_default_device`, `to_device` and `DeviceDataLoader` to leverage a GPU if available, by moving the input data and model parameters to the appropriate device.
- We were able to use the exact same training loop: the `fit` function we had define earlier to train out model and evaluate it using the validation dataset.

There's a lot of scope to experiment here, and I encourage you to use the interactive nature of Jupyter to play around with the various parameters. Here are a few ideas:

- Try changing the size of the hidden layer, or add more hidden layers and see if you can achieve a higher accuracy.
- Try changing the batch size and learning rate to see if you can achieve the same accuracy in fewer epochs.
- Compare the training times on a CPU vs. GPU. Do you see a significant difference. How does it vary with the size of the dataset and the size of the model (no. of weights and parameters)?
- Try building a model for a different dataset, such as the [CIFAR10 or CIFAR100 datasets](#).

Here are some references for further reading:

- [A visual proof that neural networks can compute any function](#), also known as the Universal Approximation Theorem.
- [But what is a neural network?](#) - A visual and intuitive introduction to what neural networks are and what the intermediate layers represent
- [Stanford CS229 Lecture notes on Backpropagation](#) - for a more mathematical treatment of how gradients are calculated and weights are updated for neural networks with multiple layers.

In [ ]: