

# Additional Materials: Convolutional Neural Networks

Intro: [https://www.youtube.com/watch?v=38ExGpdyvJI&feature=emb\\_logo](https://www.youtube.com/watch?v=38ExGpdyvJI&feature=emb_logo)

Applications: [https://www.youtube.com/watch?v=HrYNL\\_1SV2Y&feature=emb\\_logo](https://www.youtube.com/watch?v=HrYNL_1SV2Y&feature=emb_logo)

## Optional Resources

- Read about the [WaveNet](#) model.
  - Why train an A.I. to talk, when you can train it to sing ;) In April 2017, researchers used a variant of the WaveNet model to generate songs. The original paper and demo can be found [here](#).
- Learn about CNNs [for text classification](#).
  - You might like to sign up for the author's [Deep Learning Newsletter](#)!
- Read about Facebook's [novel CNN approach](#) for language translation that achieves state-of-the-art accuracy at nine times the speed of RNN models.
- Play [Atari games](#) with a CNN and reinforcement learning. You can [download](#) the code that comes with this paper.
  - If you would like to play around with some beginner code (for deep reinforcement learning), you're encouraged to check out Andrej Karpathy's [post](#).
- Play [pictionary](#) with a CNN!
  - Also check out all of the other cool implementations on the [A.I. Experiments](#) website. Be sure not to miss [AutoDraw](#)!
- Read more about [AlphaGo](#).
  - Check out [this article](#), which asks the question: *If mastering Go "requires human intuition," what is it like to have a piece of one's humanity challenged?*
- Check out these *really cool* videos with drones that are powered by CNNs.
  - Here's an interview with a startup - [Intelligent Flying Machines \(IFM\)](#).
  - Outdoor autonomous navigation is typically accomplished through the use of the [global positioning system \(GPS\)](#), but here's a demo with a CNN-powered [autonomous drone](#).
- If you're excited about using CNNs in self-driving cars, you're encouraged to check out:
  - our [Self-Driving Car Engineer Nanodegree](#), where we classify signs in the [German Traffic Sign](#) dataset in [this project](#).

- our [Machine Learning Engineer Nanodegree](#), where we classify house numbers from the [Street View House Numbers](#) dataset in [this project](#).
- this [series of blog posts](#) that details how to train a CNN in Python to produce a self-driving A.I. to play Grand Theft Auto V.
- Check out some additional applications not mentioned in the video.
  - Some of the world's most famous paintings have been [turned into 3D](#) for the visually impaired. Although the article does not mention *how* this was done, we note that it is possible to use a CNN to [predict depth](#) from a single image.
  - Check out [this research](#) that uses CNNs to localize breast cancer.
  - CNNs are used to [save endangered species](#)!
  - An app called [FaceApp](#) uses a CNN to make you smile in a picture or change genders.

Outline: [https://www.youtube.com/watch?v=77LzWE1qQrc&feature=emb\\_logo](https://www.youtube.com/watch?v=77LzWE1qQrc&feature=emb_logo)

### What is a feature?

I've found that a helpful way to think about what a **feature** is, is to think about what we are visually drawn to when we first see an object and when we identify different objects. For example, what do we look at to distinguish a cat and a dog? The shape of the eyes, the size, and how they move are just a couple of examples of visual features.

As another example, say we see a person walking toward us and we want to see if it's someone we know; we may look at their face, and even further their general shape, eyes (and even color of their eyes). The distinct shape of a person and their eye color are great examples of distinguishing features!

Next, we'll see that features like these can be measured, and represented as numerical data, by a machine.

MNIST DATA: [https://www.youtube.com/watch?v=a7bvIGZpcnk&feature=emb\\_logo](https://www.youtube.com/watch?v=a7bvIGZpcnk&feature=emb_logo)

### MNIST Data

The MNIST database is arguably the most famous database in the field of deep learning! Check out [this figure](#) that shows datasets referenced over time in [NIPS](#) papers.

### How Computers Interpret Images:

[https://www.youtube.com/watch?v=mEPfoM68Fx4&feature=emb\\_logo](https://www.youtube.com/watch?v=mEPfoM68Fx4&feature=emb_logo)

### MLP Structure & Class Scores

[https://www.youtube.com/watch?v=fP0Odiai8sk&feature=emb\\_logo](https://www.youtube.com/watch?v=fP0Odiai8sk&feature=emb_logo)

### Do Your Research

[https://www.youtube.com/watch?v=CR4JeAn1fgk&feature=emb\\_logo](https://www.youtube.com/watch?v=CR4JeAn1fgk&feature=emb_logo)

## Loss & Optimization

[https://www.youtube.com/watch?v=BmPDtSXv18w&feature=emb\\_logo](https://www.youtube.com/watch?v=BmPDtSXv18w&feature=emb_logo)

## Training the Network

[https://www.youtube.com/watch?v=904bfqibcCw&feature=emb\\_logo](https://www.youtube.com/watch?v=904bfqibcCw&feature=emb_logo)

## Cross-Entropy Loss

In the [PyTorch documentation](#), you can see that the cross entropy loss function actually involves two steps:

- It first applies a softmax function to any output it sees
- Then applies [NLLLoss](#); negative log likelihood loss

Then it returns the average loss over a batch of data. Since it applies a softmax function, we *do not* have to specify that in the `forward` function of our model definition, but we could do this another way.

### Another approach

We could separate the softmax and NLLLoss steps.

- In the `forward` function of our model, we would *explicitly* apply a softmax activation function to the output, `x`.

```
...  
...  
# a softmax layer to convert 10 outputs into a distribution of class probabilities  
x = F.log_softmax(x, dim=1)  
  
return x
```

- Then, when defining our loss criterion, we would apply `NLLLoss`

```
# cross entropy loss combines softmax and nn.NLLLoss() in one single class
```

```
# here, we've separated them
criterion = nn.NLLLoss()
```

This separates the usual `criterion = nn.CrossEntropy()` into two steps: softmax and NLLLoss, and is a useful approach should you want the output of a model to be class *probabilities* rather than class scores.

## Notebook: MLP Classification

Now, you're ready to define and train an MLP in PyTorch. As you follow along this lesson, you are encouraged to open the referenced Jupyter notebooks. We will present a solution to you, but please try creating your own deep learning models! Much of the value in this experience will come from experimenting with the code, in your own way.

To open this notebook, you have two options:

- Go to the next page in the classroom (recommended).
- Clone the repo from [Github](#) and open the notebook ***mnist\_mlp\_exercise.ipynb*** in the ***convolutional-neural-networks > mnist-mlp*** folder. You can either download the repository with `git clone https://github.com/udacity/deep-learning-v2-pytorch.git`, or download it as an archive file from [this link](#).

## Instructions

- Define an MLP model for classifying MNIST images
- Train it for some number of epochs and test your model to see how well it generalizes and measure its accuracy.

This is a self-assessed lab. If you need any help or want to check your answers, feel free to check out the solutions notebook in the same folder, or by clicking [here](#).

### One Solution

[https://www.youtube.com/watch?v=7q37WPjQhDA&feature=emb\\_logo](https://www.youtube.com/watch?v=7q37WPjQhDA&feature=emb_logo)

## `model.eval()`

There is an omission in the above code: including `model.eval()` !

`model.eval()` will set all the layers in your model to evaluation mode. This affects layers like dropout layers that turn "off" nodes during training with some probability, but should allow every node to be "on" for evaluation. So, you should set your model to evaluation mode **before testing or validating your model** and set it to `model.train()` (training mode) only during the training loop.

This is reflected in the previous notebook code and in our [Github repository](#).

## Optional Resources

- Check out the [first research paper](#) to propose dropout as a technique for overfitting.
- If you'd like more information on activation functions, check out this [website](#).

ModelValidation:

[https://www.youtube.com/watch?v=b5934VsV3SA&feature=emb\\_logo](https://www.youtube.com/watch?v=b5934VsV3SA&feature=emb_logo)

Validation Loss:

[https://www.youtube.com/watch?v=uGPP\\_-pbBsc&feature=emb\\_logo](https://www.youtube.com/watch?v=uGPP_-pbBsc&feature=emb_logo)

## Validation Code

You can take a look at the complete validation code in the previous notebook directory, or, directly in the [Github repository](#).

## Validation Set: Takeaways

We create a validation set to

1. Measure how well a model generalizes, during training
2. Tell us when to stop training a model; when the validation loss stops decreasing (and especially when the validation loss starts increasing and the training loss is still decreasing)

MLPvsCNN:

[https://www.youtube.com/watch?v=Q7CR3cCOTJQ&feature=emb\\_logo](https://www.youtube.com/watch?v=Q7CR3cCOTJQ&feature=emb_logo)

Local Connectivity

[https://www.youtube.com/watch?v=z9wiDg0w-Dc&feature=emb\\_logo](https://www.youtube.com/watch?v=z9wiDg0w-Dc&feature=emb_logo)

Filters and the Convolutional Layer

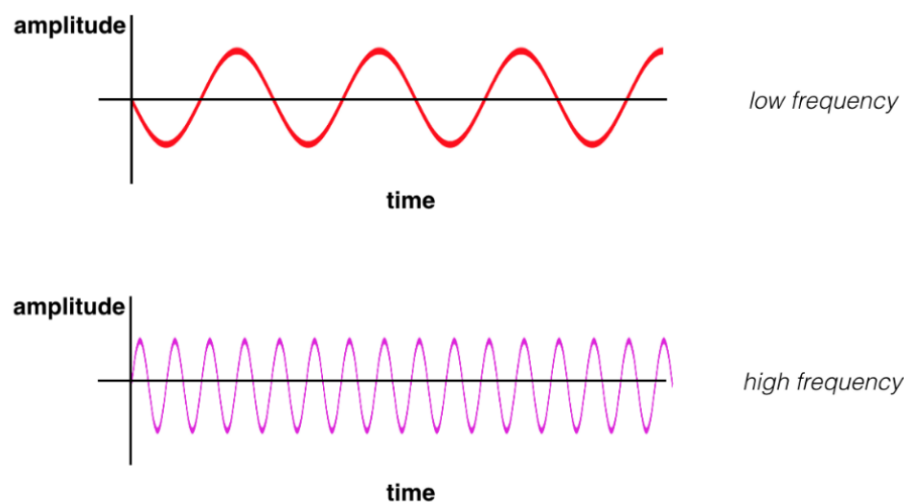
[https://www.youtube.com/watch?v=x\\_dhnhUzFNo&feature=emb\\_logo](https://www.youtube.com/watch?v=x_dhnhUzFNo&feature=emb_logo)

Filters & Edges

[https://www.youtube.com/watch?time\\_continue=5&v=hfgNqcEU6ul&feature=emb\\_logo](https://www.youtube.com/watch?time_continue=5&v=hfgNqcEU6ul&feature=emb_logo)

## Frequency in images

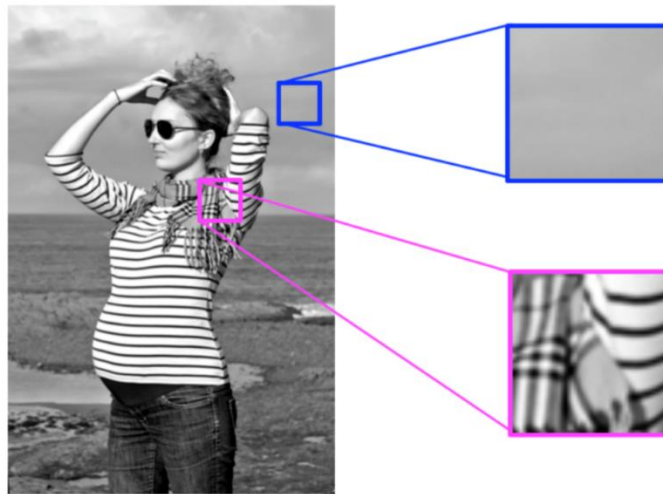
We have an intuition of what frequency means when it comes to sound. High-frequency is a high pitched noise, like a bird chirp or violin. And low frequency sounds are low pitch, like a deep voice or a bass drum. For sound, frequency actually refers to how fast a sound wave is oscillating; oscillations are usually measured in cycles/s (Hz), and high pitches and made by high-frequency waves. Examples of low and high-frequency sound waves are pictured below. On the y-axis is amplitude, which is a measure of sound pressure that corresponds to the perceived loudness of a sound, and on the x-axis is time.



(Top image) a low frequency sound wave (bottom) a high frequency sound wave.

## High and low frequency

Similarly, frequency in images is a **rate of change**. But, what does it mean for an image to change? Well, images change in space, and a high frequency image is one where the intensity changes a lot. And the level of brightness changes quickly from one pixel to the next. A low frequency image may be one that is relatively uniform in brightness or changes very slowly. This is easiest to see in an example.



High and low frequency image patterns.

Most images have both high-frequency and low-frequency components. In the image above, on the scarf and striped shirt, we have a high-frequency image pattern; this part changes very rapidly from one brightness to another. Higher up in this same image, we see parts of the sky and background that change very gradually, which is considered a smooth, low-frequency pattern.

**High-frequency components also correspond to the edges of objects in images,** which can help us classify those objects.

## High-pass Filters

[https://www.youtube.com/watch?time\\_continue=73&v=OpcFn\\_H2V-Q&feature=emb\\_logo](https://www.youtube.com/watch?time_continue=73&v=OpcFn_H2V-Q&feature=emb_logo)

## Edge Handling

Kernel convolution relies on centering a pixel and looking at its surrounding neighbors. So, what do you do if there are no surrounding pixels like on an image corner or edge? Well, there are a number of ways to process the edges, which are listed below. It's most common to use padding, cropping, or extension. In extension, the border pixels of an image are copied and extended far enough to result in a filtered image of the same size as the original image.

**Extend** The nearest border pixels are conceptually extended as far as necessary to provide values for the convolution. Corner pixels are extended in 90° wedges. Other edge pixels are extended in lines.

**Padding** The image is padded with a border of 0's, black pixels.

**Crop** Any pixel in the output image which would require values from beyond the edge is skipped. This method can result in the output image being slightly smaller, with the edges having been cropped.



## Kernel convolution

Now that you know the basics of high-pass filters, let's see if you can choose the *best* one for a given task.

a)

-1	-1	-1
-1	8	-1
-1	-1	-1

b)

-1	0	1
-2	0	2
-1	0	1

c)

0	-1	0
-2	6	-2
0	-1	0

d)

-1	-2	-1
0	0	0
1	2	1

Four different kernels

### QUIZ QUESTION

Of the four kernels pictured above, which would be best for finding and enhancing **horizontal** edges and lines in an image?

☐ a

☐ b

☐ c

☒ d

## OpenCV

Before we jump into coding our own convolutional kernels/filters, I'll introduce you to a new library that will be useful to use when dealing with computer vision tasks, such as image classification: OpenCV!



OpenCV logo

OpenCV is a **computer vision and machine learning software library** that includes many common image analysis algorithms that will help us build custom, intelligent computer vision applications. To start with, this includes tools that help us process images and select areas of interest! The library is widely used in academic and industrial applications; from [their site](#), OpenCV includes an impressive list of users: *“Along with well-established companies like Google, Yahoo, Microsoft, Intel, IBM, Sony, Honda, Toyota that employ the library, there are many startups such as Applied Minds, VideoSurf, and Zeitera, that make extensive use of OpenCV.”*

So, note, how we `import cv2` in the next notebook and use it to create and apply image filters!

## Notebook: Custom Filters

The next notebook is called `custom_filters.ipynb`.

To open the notebook, you have two options:

- *Go to the next page in the classroom (recommended).*

- Clone the repo from [Github](https://github.com/udacity/deep-learning-v2-pytorch) and open the notebook **custom\_filters.ipynb** in the **convolutional-neural-networks > conv-visualization** folder. You can either download the repository with `git clone https://github.com/udacity/deep-learning-v2-pytorch.git`, or download it as an archive file from [this link](#).

## Instructions

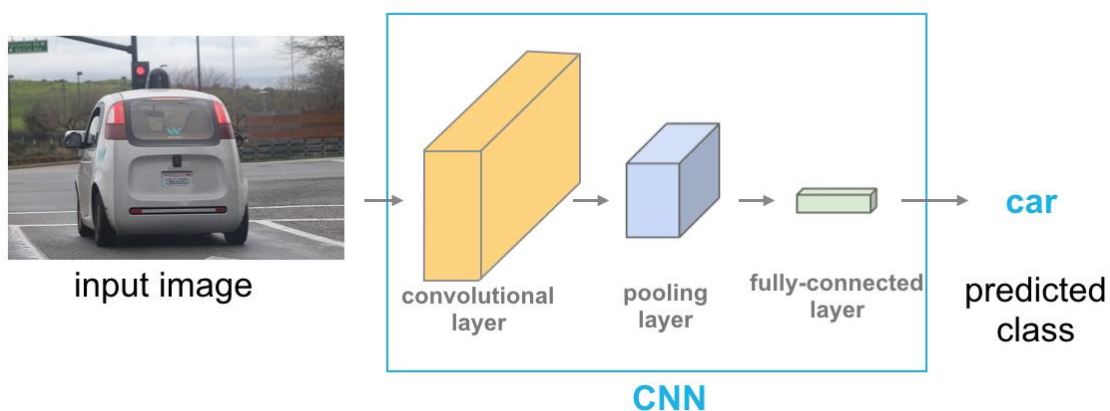
- Define your own convolutional filters and apply them to an image of a road
- See if you can define filters that detect horizontal or vertical edges

This notebook is meant to be a playground where you can try out different filter sizes and weights and see the resulting, filtered output image!

### ➤ Notebook: Finding edges

## The Importance of Filters

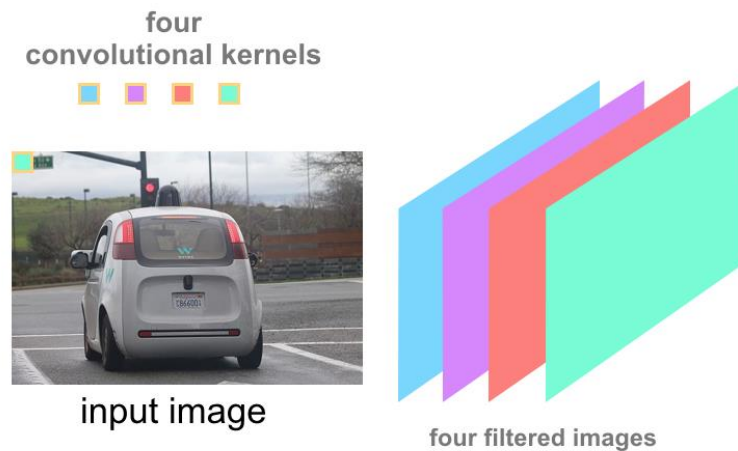
What you've just learned about different types of filters will be really important as you progress through this course, especially when you get to Convolutional Neural Networks (CNNs). CNNs are a kind of deep learning model that can learn to do things like image classification and object recognition. They keep track of spatial information and *learn* to extract features like the edges of objects in something called a **convolutional layer**. Below you'll see a simple CNN structure, made of multiple layers, below, including this "convolutional layer".



## Layers in a CNN.

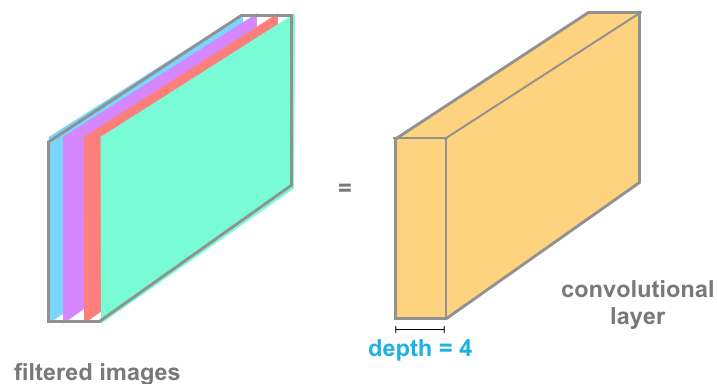
### Convolutional Layer

The convolutional layer is produced by applying a series of many different image filters, also known as convolutional kernels, to an input image.



**4 kernels = 4 filtered images.**

In the example shown, 4 different filters produce 4 differently filtered output images. When we stack these images, we form a complete convolutional layer with a depth of 4!



**A convolutional layer.**

## Image Augmentation In Keras:

[https://www.youtube.com/watch?time\\_continue=47&v=zQnx2jZmjTA&feature=emb\\_logo](https://www.youtube.com/watch?time_continue=47&v=zQnx2jZmjTA&feature=emb_logo)

Augmentation Using Transformations

### Augmentation Code

You can take a look at the complete augmentation code in the previous notebook directory, or, directly in the [Github repository](#).

[https://www.youtube.com/watch?time\\_continue=30&v=J\\_gjHVt9pVw&feature=emb\\_logo](https://www.youtube.com/watch?time_continue=30&v=J_gjHVt9pVw&feature=emb_logo)

GROUND BREAKING ARCHITECTURE CNN:

[https://www.youtube.com/watch?v=GdYOqihgb2k&feature=emb\\_logo](https://www.youtube.com/watch?v=GdYOqihgb2k&feature=emb_logo)

## Optional Resources

- Check out the [AlexNet](#) paper!
- Read more about [VGGNet](#) here.
- The [ResNet](#) paper can be found here.
- Here's the Keras [documentation](#) for accessing some famous CNN architectures.
- Read this [detailed treatment](#) of the vanishing gradients problem.
- Here's a GitHub [repository](#) containing benchmarks for different CNN architectures.
- Visit the [ImageNet Large Scale Visual Recognition Competition \(ILSVRC\)](#) website.

# Transfer Learning

Def:

[https://www.youtube.com/watch?time\\_continue=1&v=yfPEROi3SPU&feature=emb\\_logo](https://www.youtube.com/watch?time_continue=1&v=yfPEROi3SPU&feature=emb_logo)

Useful Layers:

[https://www.youtube.com/watch?v=kn4BN7z3UGQ&feature=emb\\_logo](https://www.youtube.com/watch?v=kn4BN7z3UGQ&feature=emb_logo)

Fine-Tuning

[https://www.youtube.com/watch?v=XOyb315xYbw&feature=emb\\_logo](https://www.youtube.com/watch?v=XOyb315xYbw&feature=emb_logo)

# Transfer Learning

Transfer learning involves taking a pre-trained neural network and adapting the neural network to a new, different data set.

Depending on both:

- The size of the new data set, and
- The similarity of the new data set to the original data set

The approach for using transfer learning will be different. There are four main cases:

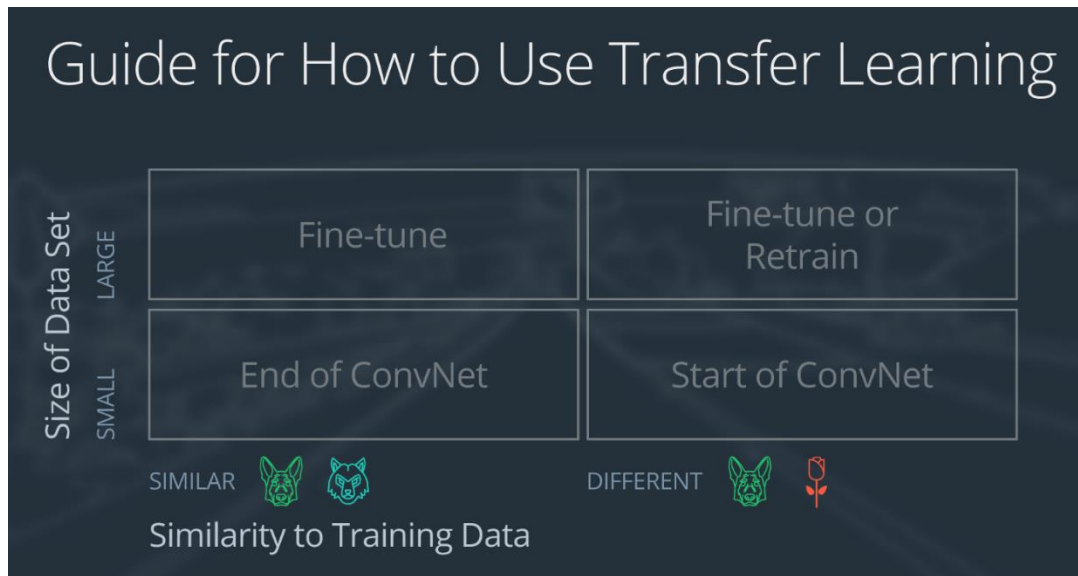
1. New data set is small, new data is similar to original training data.
2. New data set is small, new data is different from original training data.
3. New data set is large, new data is similar to original training data.
4. New data set is large, new data is different from original training data.

A large data set might have one million images. A small data could have two-thousand images. The dividing line between a large data set and small data set is somewhat subjective. Overfitting is a concern when using transfer learning with a small data set.

Images of dogs and images of wolves would be considered similar; the images would share common characteristics. A data set of flower images would be different from a data set of dog images.

Each of the four transfer learning cases has its own approach. In the following sections, we will look at each case one by one.

The graph below displays what approach is recommended for each of the four main cases.

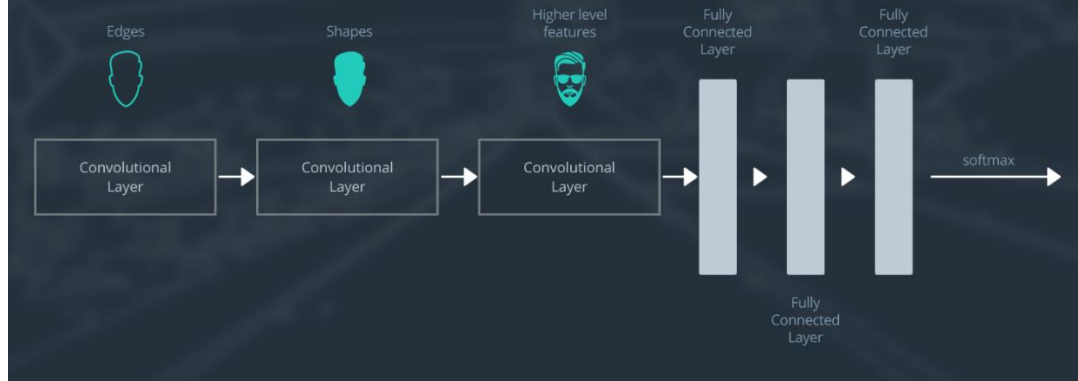


**Four cases for using transfer learning.**

## Demonstration Network

To explain how each situation works, we will start with a generic pre-trained convolutional neural network and explain how to adjust the network for each case. Our example network contains three convolutional layers and three fully connected layers:

# Pre-trained Convolutional Neural Network



**Overview of the layers of a pre-trained CNN.**

Here is an generalized overview of what the convolutional neural network does:

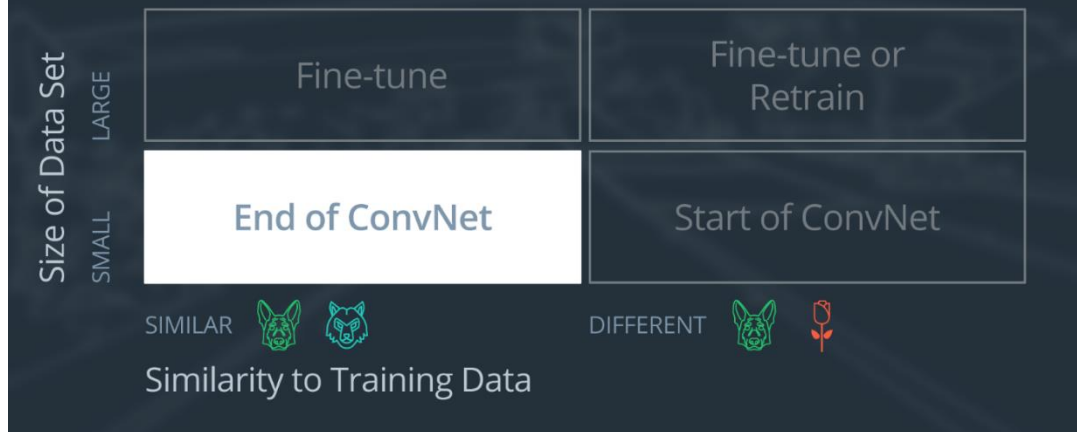
- the first layer will detect edges in the image
- the second layer will detect shapes
- the third convolutional layer detects higher level features

Each transfer learning case will use the pre-trained convolutional neural network in a different way.

## **Case 1: Small Data Set, Similar Data**



## Case: Small Data Set, Similar Data



### Case 1: small set, similar data

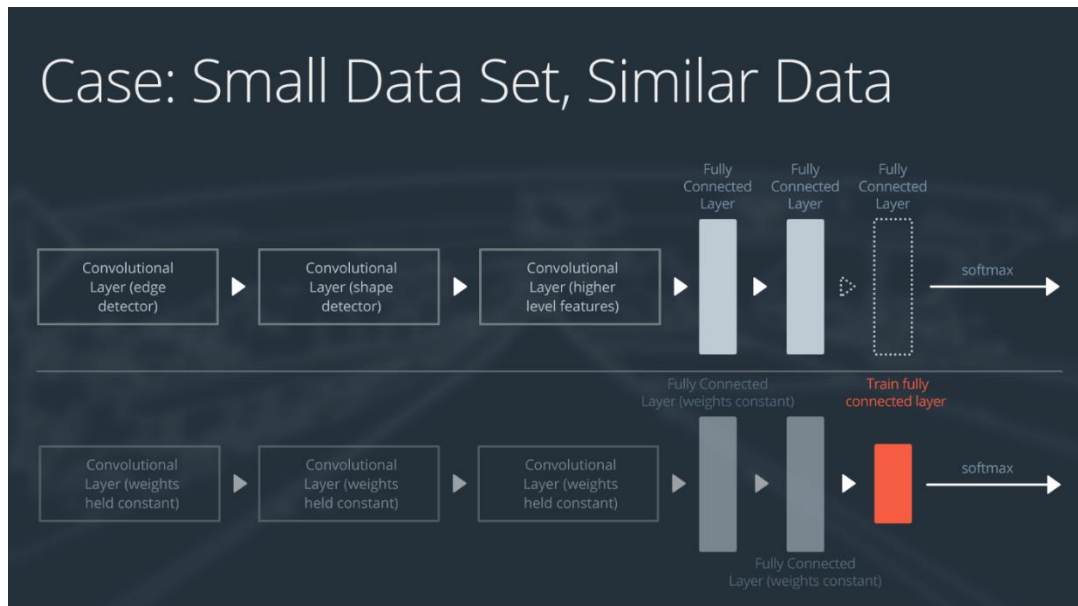
If the new data set is small and similar to the original training data:

- slice off the end of the neural network
- add a new fully connected layer that matches the number of classes in the new data set
- randomize the weights of the new fully connected layer; freeze all the weights from the pre-trained network
- train the network to update the weights of the new fully connected layer

To avoid overfitting on the small data set, the weights of the original network will be held constant rather than re-training the weights.

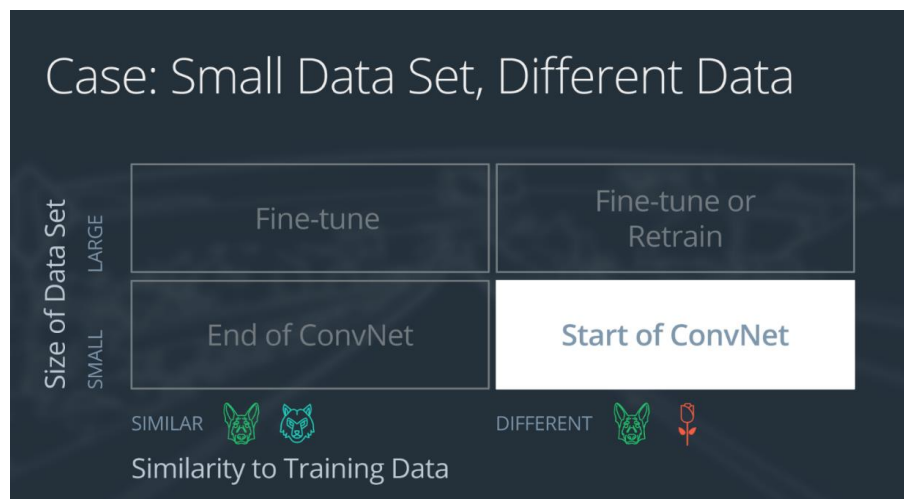
Since the data sets are similar, images from each data set will have similar higher level features. Therefore most or all of the pre-trained neural network layers already contain relevant information about the new data set and should be kept.

Here's how to visualize this approach:



**Adding and training a fully-connected layer at the end of the NN.**

## Case 2: Small Data Set, Different Data



**Case 2: small set, different data**

If the new data set is small and different from the original training data:

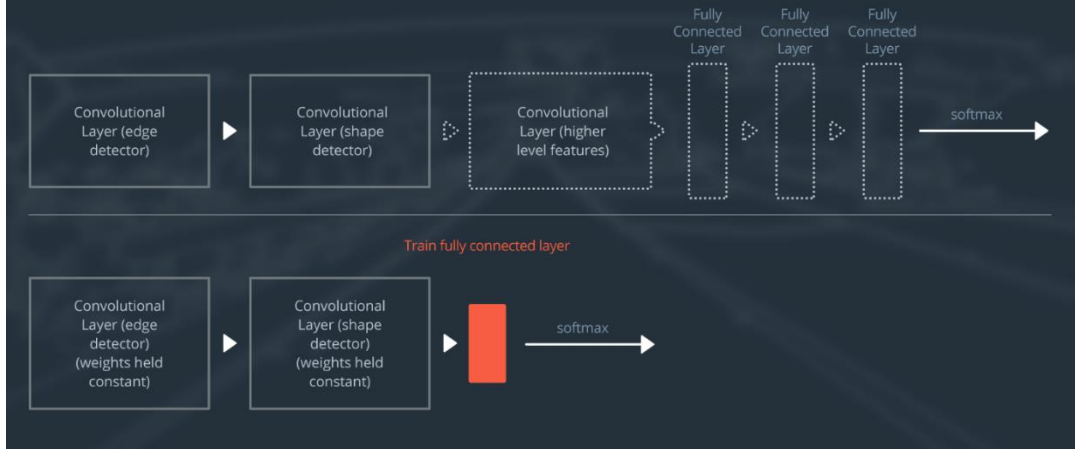
- slice off all but some of the pre-trained layers near the beginning of the network
- add to the remaining pre-trained layers a new fully connected layer that matches the number of classes in the new data set
- randomize the weights of the new fully connected layer; freeze all the weights from the pre-trained network
- train the network to update the weights of the new fully connected layer

Because the data set is small, overfitting is still a concern. To combat overfitting, the weights of the original neural network will be held constant, like in the first case.

But the original training set and the new data set do not share higher level features. In this case, the new network will only use the layers containing lower level features.

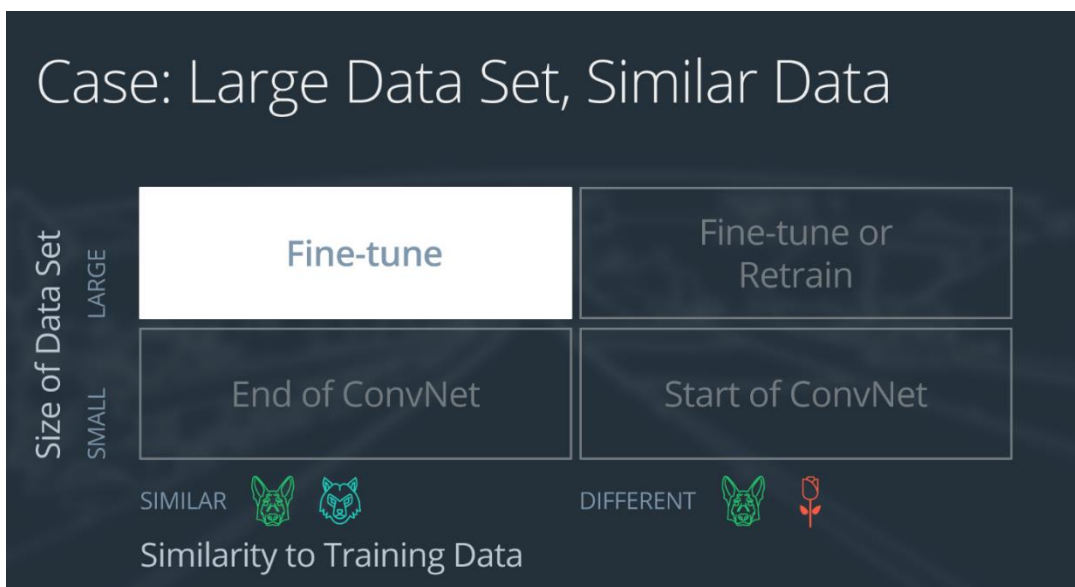
Here is how to visualize this approach:

## Case: Small Data Set, Different Data



Remove all but the starting layers of the model, and add and train a linear layer at the end.

### Case 3: Large Data Set, Similar Data



**Case 3: large data, similar to ImageNet or pre-trained set.**

If the new data set is large and similar to the original training data:

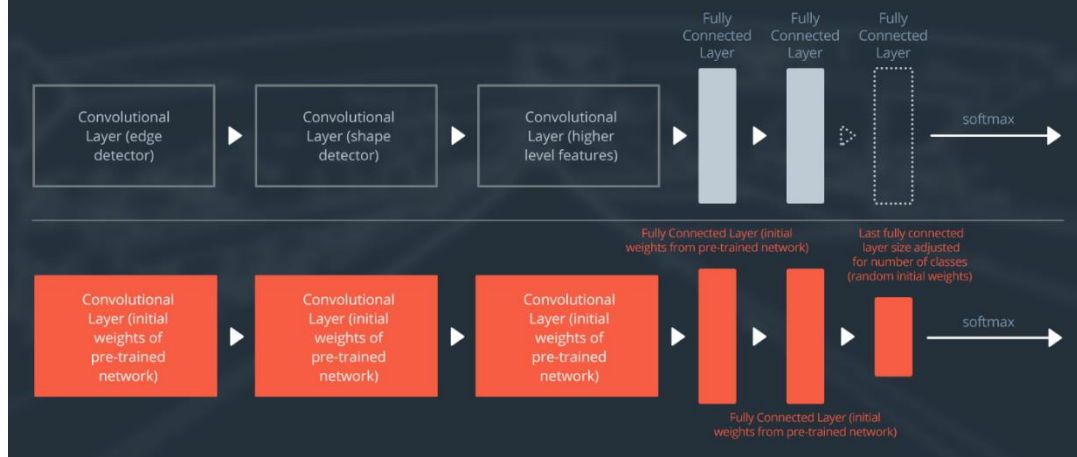
- remove the last fully connected layer and replace with a layer matching the number of classes in the new data set
- randomly initialize the weights in the new fully connected layer
- initialize the rest of the weights using the pre-trained weights
- re-train the entire neural network

Overfitting is not as much of a concern when training on a large data set; therefore, you can re-train all of the weights.

Because the original training set and the new data set share higher level features, the entire neural network is used as well.

Here is how to visualize this approach:

## Case: Large Data Set, Similar Data



**Utilizing pre-trained weights as a starting point!**

## Case 4: Large Data Set, Different Data



**Case 4: large data, different than original set**

If the new data set is large and different from the original training data:

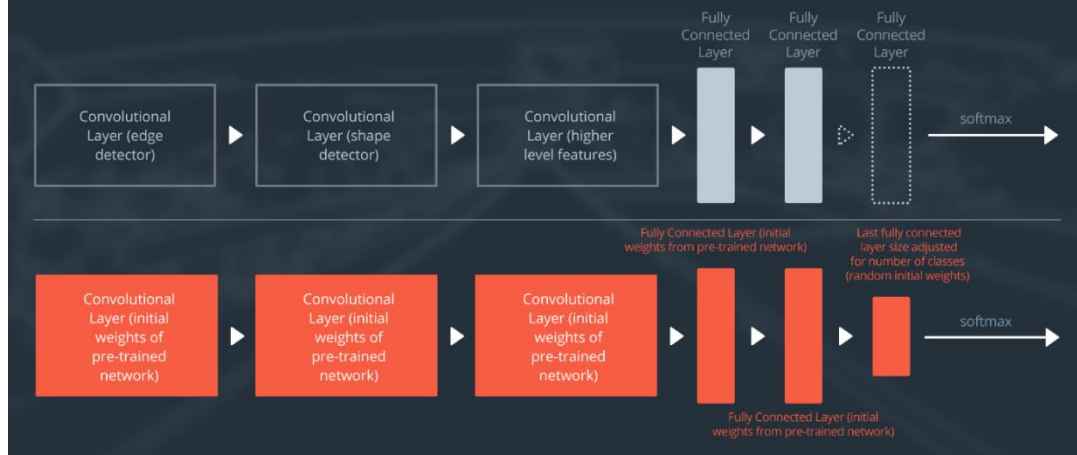
- remove the last fully connected layer and replace with a layer matching the number of classes in the new data set
- retrain the network from scratch with randomly initialized weights
- alternatively, you could just use the same strategy as the "large and similar" data case

Even though the data set is different from the training data, initializing the weights from the pre-trained network might make training faster. So this case is exactly the same as the case with a large, similar data set.

If using the pre-trained network as a starting point does not produce a successful model, another option is to randomly initialize the convolutional neural network weights and train the network from scratch.

Here is how to visualize this approach:

## Case: Large Data Set, Different Data



**Fine-tune or retrain entire network.**

### Optional Resources

- Check out this [research paper](#) that systematically analyzes the transferability of features learned in pre-trained CNNs.
- Read the [Nature publication](#) detailing Sebastian Thrun's cancer-detecting CNN!

VGG Classifier

[https://www.youtube.com/watch?time\\_continue=9&v=fOiQFXItYe4&feature=emb\\_logo](https://www.youtube.com/watch?time_continue=9&v=fOiQFXItYe4&feature=emb_logo)

### Notebook: Transfer Learning

Now, you're ready to use transfer learning on a new task!



**It's suggested that you open the notebook in a new, working tab and continue working on it as you go through the instructional videos in this tab.** This way you can toggle between learning new skills and coding/applying new skills.

To open this notebook, you have two options:

- Go to the next page in the classroom (recommended).
- Clone the repo from [Github](#) and open the notebook **Transfer\_Learning\_Exercise.ipynb** in the **transfer-learning** folder.

You can either download the repository with git clone

<https://github.com/udacity/deep-learning-v2-pytorch.git>, or download it as an archive file from [this link](#).

### Instructions

- Load in a pre-trained VGG Net
- Freeze the weights in selected layers and add a new, linear layer of your own design
- Train the modified model for a couple epochs and test its performance

This is a self-assessed lab. If you need any help or want to check your answers, feel free to check out the solutions notebook in the same folder, or by clicking [here](#).

### GPU Workspaces

The next workspace is **GPU-enabled**, which means you can select to train on a GPU instance. The recommendation is this:

- Load in data, test functions and models (checking parameters and doing a short training loop) while in CPU (non-enabled) mode
- When you're ready to extensively train and test your model, **enable** GPU to quickly train the model!

All models and data they see as input will have to be moved to the GPU device, so take note of the relevant movement code in the model creation and training process.

## Freezing parameters

To freeze any parameters, you can use the variable `requires_grad`. By default this is set to True. To freeze existing parameters you can loop through each one and set `param.requires_grad = False`.

Above, there is a **small typo**, which leaves out the `s` in `requires_grad`. This has been fixed in the exercise notebooks and our Github repo.

[https://www.youtube.com/watch?time\\_continue=4&v=ssNIX\\_2QfMQ&feature=emb\\_logo](https://www.youtube.com/watch?time_continue=4&v=ssNIX_2QfMQ&feature=emb_logo)

Training a Classifier:

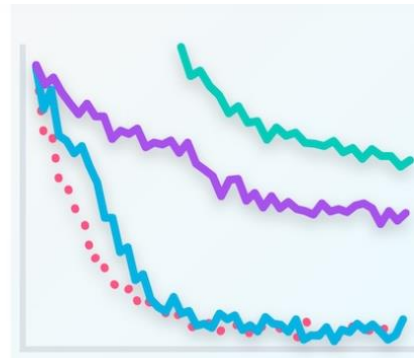
[https://www.youtube.com/watch?time\\_continue=8&v=4LniBMFI53g&feature=emb\\_logo](https://www.youtube.com/watch?time_continue=8&v=4LniBMFI53g&feature=emb_logo)

### LESSON 5

#### Weight Initialization

In this lesson, you'll learn how to find good initial weights for a neural network. Having good initial weights can place the neural network closer to the optimal solution.

CONTINUE →



[https://www.youtube.com/watch?time\\_continue=3&v=Ehc60si91Wg&feature=emb\\_logo](https://www.youtube.com/watch?time_continue=3&v=Ehc60si91Wg&feature=emb_logo)

Constant Weights

[https://www.youtube.com/watch?v=zR4fECgeZ7Y&feature=emb\\_logo](https://www.youtube.com/watch?v=zR4fECgeZ7Y&feature=emb_logo)

Random Uniform

[https://www.youtube.com/watch?v=FacdlomrLlw&feature=emb\\_logo](https://www.youtube.com/watch?v=FacdlomrLlw&feature=emb_logo)

General Rule

[https://www.youtube.com/watch?v=YKe9iOUMmsl&feature=emb\\_logo](https://www.youtube.com/watch?v=YKe9iOUMmsl&feature=emb_logo)

Normal Distribution

[https://www.youtube.com/watch?time\\_continue=18&v=xm43q4qD2tl&feature=emb\\_logo](https://www.youtube.com/watch?time_continue=18&v=xm43q4qD2tl&feature=emb_logo)

## Notebook: Weight Initialization

Now, you're ready to try out a weight initialization method that you define.

To open this notebook, you have two options:

- Go to the next page in the classroom (recommended).
- Clone the repo from [Github](#) and open the notebook ***weight\_initialization\_exercise.ipynb*** in the *\*weight-initialization* folder. You can either download the repository with `git clone https://github.com/udacity/deep-learning-v2-pytorch.git`, or download it as an archive file from [this link](#).

## Instructions

- Load in the FashionMNIST data
  - Define a function to initialize the weights of your model, taking values from a normal distribution
  - See how a model *without any explicit* weight initialization performs
- This is a self-assessed lab. If you need any help or want to check your answers, feel free to check out the solutions notebook in the same folder, or by clicking [here](#).

Solution:

[https://www.youtube.com/watch?v=xln8XLbR1LM&feature=emb\\_logo](https://www.youtube.com/watch?v=xln8XLbR1LM&feature=emb_logo)

## Additional Material

New techniques for dealing with weights are discovered every few years. We've provided the most popular papers in this field over the years.

- [Understanding the difficulty of training deep feedforward neural networks](#)

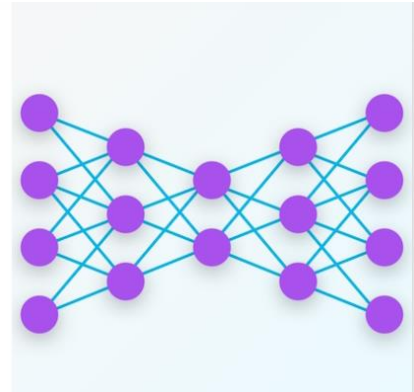
- [Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification](#)
- [Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift](#)

#### LESSON 6

##### Autoencoders

Autoencoders are neural networks used for data compression, image denoising, and dimensionality reduction. Here, you'll build autoencoders using PyTorch.

CONTINUE →



[https://www.youtube.com/watch?v=a5zHMWOq0fc&feature=emb\\_logo](https://www.youtube.com/watch?v=a5zHMWOq0fc&feature=emb_logo)

#### A Linear Autoencoder

[https://www.youtube.com/watch?time\\_continue=11&v=KbmfyDNxL5U&feature=emb\\_logo](https://www.youtube.com/watch?time_continue=11&v=KbmfyDNxL5U&feature=emb_logo)

## Notebook: Linear Autoencoder

It's suggested that you open the notebook in a new, working tab and continue working on it as you go through the instructional videos in this tab. This way you can toggle between learning new skills and coding/applying new skills.

To open this notebook, you have two options:

- Go to the next page in the classroom (recommended).
- Clone the repo from [Github](#) and open the notebook

**Simple\_Autoencoder\_Exercise.ipynb** in the **autoencoder > linear-autoencoder**

folder. You can either download the repository with `git clone https://github.com/udacity/deep-learning-v2-pytorch.git`, or download it as an archive file from [this link](#).

# Instructions

- Define and train a linear autoencoder

This is a self-assessed lab. If you need any help or want to check your answers, feel free to check out the solutions notebook in the same folder, or by clicking [here](#).

Solution:

[https://www.youtube.com/watch?time\\_continue=3&v=Jh3mbomqpw8&feature=emb\\_logo](https://www.youtube.com/watch?time_continue=3&v=Jh3mbomqpw8&feature=emb_logo)

Learnable Upsampling

[https://www.youtube.com/watch?time\\_continue=18&v=KjztLwPksj8&feature=emb\\_logo](https://www.youtube.com/watch?time_continue=18&v=KjztLwPksj8&feature=emb_logo)

Transpose Convolutions

[https://www.youtube.com/watch?time\\_continue=32&v=hnnLAC1Q0zg&feature=emb\\_logo](https://www.youtube.com/watch?time_continue=32&v=hnnLAC1Q0zg&feature=emb_logo)

Convolutional Autoencoder

[https://www.youtube.com/watch?time\\_continue=28&v=QCA8QeZeDW8&feature=emb\\_logo](https://www.youtube.com/watch?time_continue=28&v=QCA8QeZeDW8&feature=emb_logo)

## Notebook: Convolutional Autoencoder

It's suggested that you open the notebook in a new, working tab and continue working on it as you go through the instructional videos in this tab. This way you can toggle between learning new skills and coding/applying new skills.

To open this notebook, you have two options:

- Go to the next page in the classroom (recommended).
- Clone the repo from [Github](#) and open the notebook

**Convolutional\_Autoencoder\_Exercise.ipynb** in the **autoencoder > convolutional-autoencoder** folder. You can either download the repository with `git clone https://github.com/udacity/deep-learning-v2-pytorch.git`, or download it as an archive file from [this link](#).

## Instructions

- Define and train a convolutional autoencoder

This is a self-assessed lab. If you need any help or want to check your answers, feel free to check out the solutions notebook in the same folder, or by clicking [here](#).

### Exercise: GPU Workspaces

***You do not need to enable GPU to complete this exercise.** This is left as an optional exercise; you will be responsible for moving your models and data to GPU, should you choose to enable it.*

The next workspace is **GPU-enabled**, which means you can select to train on a GPU instance. The recommendation is this:

- Load in data, test functions and models (checking parameters and doing a short training loop) while in CPU (non-enabled) mode
- When you're ready to extensively train and test your model, **enable** GPU to quickly train the model!

All models and data they see as input will have to be moved to the GPU device, so take note of the relevant movement code in the model creation and training process.

Convolutional Solution

[https://www.youtube.com/watch?time\\_continue=11&v=2\\_Yw9LLomCo&feature=emb\\_logo](https://www.youtube.com/watch?time_continue=11&v=2_Yw9LLomCo&feature=emb_logo)

Upsampling & Denoising

[https://www.youtube.com/watch?time\\_continue=22&v=XX63da4EPN0&feature=emb\\_logo](https://www.youtube.com/watch?time_continue=22&v=XX63da4EPN0&feature=emb_logo)

De-noising

[https://www.youtube.com/watch?time\\_continue=7&v=RlfEhKev24I&feature=emb\\_logo](https://www.youtube.com/watch?time_continue=7&v=RlfEhKev24I&feature=emb_logo)

### Notebook: De-noising Autoencoder

Try defining and training an autoencoder for denoising images!

To open this notebook, you have two options:

- Go to the next page in the classroom (recommended).
- Clone the repo from [Github](#) and open the notebook

***Denoising\_Autoencoder\_Exercise.ipynb*** in the **autoencoder > denoising-autoencoder** folder. You can either download the repository with git clone

<https://github.com/udacity/deep-learning-v2-pytorch.git>, or download it as an archive file from [this link](#).

## Instructions

- Define and train a convolutional autoencoder
- Add more/deeper layers to create a successful de-noiser

This is a self-assessed lab. If you need any help or want to check your answers, feel free to check out *one kind of solution* in the following notebook, or by clicking [here](#).

## Exercise: GPU Workspaces

***You do not need to enable GPU to complete this exercise.*** This is left as an optional exercise; you will be responsible for moving your models and data to GPU, should you choose to enable it.

The next workspace is **GPU-enabled**, which means you can select to train on a GPU instance. The recommendation is this:

- Load in and test models while in CPU (non-enabled) mode
- When you're ready to extensively train and test your model, you'll have to add GPU functionality (in this case, the code *is not* provided for you)
- Once you've moved your model and data to GPU, you can **enable** GPU to quickly train the model!

All models and data they see as input will have to be moved to the GPU device, so take note of the relevant movement code in the model creation and training process.

## Extra Curriculum:

### LESSON 1: Introduction to NLP:

NLP and Pipelines:

[https://www.youtube.com/watch?v=UQBxJzoCp-I&feature=emb\\_logo](https://www.youtube.com/watch?v=UQBxJzoCp-I&feature=emb_logo)

How NLP Pipelines Work:

[https://www.youtube.com/watch?time\\_continue=1&v=vJx6oKlu\\_MM&feature=emb\\_logo](https://www.youtube.com/watch?time_continue=1&v=vJx6oKlu_MM&feature=emb_logo)

Text Processing:

[https://www.youtube.com/watch?time\\_continue=16&v=pqheVyctkNQ&feature=emb\\_logo](https://www.youtube.com/watch?time_continue=16&v=pqheVyctkNQ&feature=emb_logo)

Feature Extraction:

[https://www.youtube.com/watch?time\\_continue=14&v=Bd6TJB8eVLQ&feature=emb\\_logo](https://www.youtube.com/watch?time_continue=14&v=Bd6TJB8eVLQ&feature=emb_logo)

Bag Of Words:

[https://www.youtube.com/watch?v=A7M1z8yLI0w&feature=emb\\_logo](https://www.youtube.com/watch?v=A7M1z8yLI0w&feature=emb_logo)

TF-IDF:

[https://www.youtube.com/watch?v=XZBiBIRcACE&feature=emb\\_logo](https://www.youtube.com/watch?v=XZBiBIRcACE&feature=emb_logo)

One-Hot Encoding:

[https://www.youtube.com/watch?time\\_continue=24&v=a0j1CDXFYZI&feature=emb\\_logo](https://www.youtube.com/watch?time_continue=24&v=a0j1CDXFYZI&feature=emb_logo)

Word Embeddings:

[https://www.youtube.com/watch?time\\_continue=15&v=4mM\\_S9L2\\_JQ&feature=emb\\_logo](https://www.youtube.com/watch?time_continue=15&v=4mM_S9L2_JQ&feature=emb_logo)

Word2Vec:

[https://www.youtube.com/watch?time\\_continue=15&v=4mM\\_S9L2\\_JQ&feature=emb\\_logo](https://www.youtube.com/watch?time_continue=15&v=4mM_S9L2_JQ&feature=emb_logo)

GloVe:

[https://www.youtube.com/watch?v=KK3PMIln8o&feature=emb\\_logo](https://www.youtube.com/watch?v=KK3PMIln8o&feature=emb_logo)

Embeddings for Deep Learning:

[https://www.youtube.com/watch?time\\_continue=30&v=gj8u1KG0H2w&feature=emb\\_logo](https://www.youtube.com/watch?time_continue=30&v=gj8u1KG0H2w&feature=emb_logo)

Modeling:

[https://www.youtube.com/watch?time\\_continue=16&v=P4w\\_2rkxBvE&feature=emb\\_logo](https://www.youtube.com/watch?time_continue=16&v=P4w_2rkxBvE&feature=emb_logo)

## LESSON 2: Implementation of RNN & LSTM

Implementing RNNs:

[https://www.youtube.com/watch?v=BHoiwB61ays&feature=emb\\_logo](https://www.youtube.com/watch?v=BHoiwB61ays&feature=emb_logo)

### Code Walkthrough & Repository

The below video is a walkthrough of code that you can find in our public Github repository, if you navigate to recurrent-neural-networks > time-series and the Simple\_RNN.ipynb notebook. Feel free to go through this code on your own, locally.

This example is meant to give you an idea of how PyTorch represents RNNs and how you might represent memory in code. Later, you'll be given more complex exercise and solution notebooks, in-classroom.



[https://www.youtube.com/watch?v=xV5jHLFfJbQ&feature=emb\\_logo](https://www.youtube.com/watch?v=xV5jHLFfJbQ&feature=emb_logo)

## Recurrent Layers

Here is the documentation for the main types of [recurrent layers in PyTorch](#). Take a look and read about the three main types: RNN, LSTM, and GRU. 3<sup>RD</sup> ANS BELOW

[https://www.youtube.com/watch?time\\_continue=24&v=sx7T\\_KP5v9I&feature=emb\\_logo](https://www.youtube.com/watch?time_continue=24&v=sx7T_KP5v9I&feature=emb_logo)

### QUIZ QUESTION

Say you've defined a GRU layer with `input_size = 100`, `hidden_size = 20`, and `num_layers=1`. What will the dimensions of the hidden state be if you're passing in data, batch first, in batches of 3 sequences at a time?

- ☐ (1, 1, 20)
- ☐ (1, 1, 100)
- ☐ (1, 3, 20)
- ☐ (1, 3, 100)
- ☐ (3, 1, 20)

Yes! The hidden state should have dimensions:

`(num_layers, batch_size, hidden_dim)`.

## Character-wise RNNs

[https://www.youtube.com/watch?time\\_continue=15&v=dXI3eWCGLdU&feature=emb\\_logo](https://www.youtube.com/watch?time_continue=15&v=dXI3eWCGLdU&feature=emb_logo)

## Sequence Batching

[https://www.youtube.com/watch?time\\_continue=1&v=Z4OiyU0Cldg&feature=emb\\_logo](https://www.youtube.com/watch?time_continue=1&v=Z4OiyU0Cldg&feature=emb_logo)

## Notebook: Character-Level RNN

Now you have all the information you need to implement an RNN of our own. The next few videos will be all about character-level text prediction with an LSTM!

**It's suggested that you open the notebook in a new, working tab and continue working on it as you go through the instructional videos in this tab.** This way you can toggle between learning new skills and coding/applying new skills.

To open this notebook, you have two options:

- Go to the next page in the classroom (recommended).
- Clone the repo from [Github](#) and open the notebook **Character\_Level\_RNN\_Exercise.ipynb** in the **recurrent-neural-networks > char-rnn** folder. You can either download the repository with git clone <https://github.com/udacity/deep-learning-v2-pytorch.git>, or download it as an archive file from [this link](#).

## Instructions

- Load in text data
- Pre-process that data, encoding characters as integers and creating one-hot input vectors
- Define an RNN that predicts the *next* character when given an input sequence
- Train the RNN and use it to generate *new* text

This is a self-assessed lab. If you need any help or want to check your answers, feel free to check out the solutions notebook in the same folder, or by clicking [here](#).

## GPU Workspaces

The next workspace is **GPU-enabled**, which means you can select to train on a GPU instance. The recommendation is this:

- Load in data, test functions and models (checking parameters and doing a short training loop) while in CPU (non-enabled) mode
- When you're ready to extensively train and test your model, **enable** GPU to quickly train the model!

All models and data they see as input will have to be moved to the GPU device, so take note of the relevant movement code in the model creation and training process.

- See the notebook 'Character level RNN'

Implementing a Char-RNN:

[https://www.youtube.com/watch?v=MMtgZXzFB10&feature=emb\\_logo](https://www.youtube.com/watch?v=MMtgZXzFB10&feature=emb_logo)

Batching Data, Solution:

[https://www.youtube.com/watch?time\\_continue=9&v=9Eg0wf3eW-k&feature=emb\\_logo](https://www.youtube.com/watch?time_continue=9&v=9Eg0wf3eW-k&feature=emb_logo)

Defining the Model:

[https://www.youtube.com/watch?v=LWzyqq4hCY&feature=emb\\_logo](https://www.youtube.com/watch?v=LWzyqq4hCY&feature=emb_logo)

Char-RNN, Solution:

[https://www.youtube.com/watch?v=ed33qePHrJM&feature=emb\\_logo](https://www.youtube.com/watch?v=ed33qePHrJM&feature=emb_logo)

## Representing Memory

You've learned that RNN's work well for sequences of data because they have a kind of memory. This memory is represented by something called the **hidden state**.

In the character-level LSTM example, each LSTM cell, in addition to accepting a character as input and generating an output character, also has some hidden state, and each cell will pass along its hidden state to the next cell.

This connection creates a kind of memory by which a series of cells can remember which characters they've just seen and use that information to inform the next prediction!

For example, if a cell has just generated the character a it likely will *not* generate another a, right after that!

### **net.eval()**

There is an omission in the above code: including `net.eval()` !

`net.eval()` will set all the layers in your model to evaluation mode. This affects layers like dropout layers that turn "off" nodes during training with some probability, but should allow every node to be "on" for evaluation. So, you should set your model to evaluation mode **before testing or validating your model**, and before, for example, sampling and making predictions about the likely next character in a given sequence. I'll set `net.train()` (training mode) only during the training loop.

This is reflected in the previous notebook code and in our [Github repository](#).

### Making Predictions:

### **Examples of RNNs**

Take a look at one of my favorite examples of RNNs making predictions based on some user-generated input dat: the [sketch-rnn by Magenta](#). This RNN takes as input a starting sketch, drawn by you, and then tries to complete your sketch using a particular model. For example, it can learn to complete a sketch of a pineapple or the mona lisa!

## **Complete Sentiment RNN**

Link to make your own sentiment analysis:

[https://github.com/udacity/deep-learning-v2-pytorch/blob/master/sentiment-rnn/Sentiment\\_RNN\\_Solution.ipynb](https://github.com/udacity/deep-learning-v2-pytorch/blob/master/sentiment-rnn/Sentiment_RNN_Solution.ipynb)

### **Consult the Solution Code**

To take a closer look at this solution, feel free to check out the solution workspace or click [here](#) to see it as a webpage.

### **Complete RNN Class**

I hope you tried out defining this model on your own and got it to work! Below, is how I completed this model.

I know I want an embedding layer, a recurrent layer, and a final, linear layer with a sigmoid applied; I defined all of those in the `__init__` function, according to passed in parameters.

```
def __init__(self, vocab_size, output_size, embedding_dim, hidden_dim, n_layers,
drop_prob=0.5):
```

```
    """
```

Initialize the model by setting up the layers.

```
"""
super(SentimentRNN, self).__init__()

self.output_size = output_size
self.n_layers = n_layers
self.hidden_dim = hidden_dim

# embedding and LSTM layers
self.embedding = nn.Embedding(vocab_size, embedding_dim)
self.lstm = nn.LSTM(embedding_dim, hidden_dim, n_layers,
                    dropout=drop_prob, batch_first=True)

# dropout layer
self.dropout = nn.Dropout(0.3)

# linear and sigmoid layers
self.fc = nn.Linear(hidden_dim, output_size)
self.sig = nn.Sigmoid()
```

### `__init__` explanation

First I have an **embedding layer**, which should take in the size of our vocabulary (our number of integer tokens) and produce an embedding of `embedding_dim` size. So, as this model trains, this is going to create an embedding lookup table that has as many rows as we have word integers, and as many columns as the embedding dimension.

Then, I have an **LSTM layer**, which takes in inputs of `embedding_dim` size. So, it's accepting embeddings as inputs, and producing an output and hidden state of a hidden size. I am also specifying a number of layers, and a dropout value, and finally, I'm setting `batch_first` to `True` because we are using `DataLoaders` to batch our data like that!

Then, the LSTM outputs are passed to a dropout layer and then a fully-connected, linear layer that will produce `output_size` number of outputs. And finally, I've defined a sigmoid layer to convert the output to a value between 0-1.

### Feedforward behavior

Moving on to the forward function, which takes in an input x and a hidden state, I am going to pass an input through these layers in sequence.

```
def forward(self, x, hidden):
```

```
    """
```

```
    Perform a forward pass of our model on some input and hidden state.
```

```
    """
```

```
    batch_size = x.size(0)
```

```
    # embeddings and lstm_out
```

```
    embeds = self.embedding(x)
```

```
    lstm_out, hidden = self.lstm(embeds, hidden)
```

```
    # stack up lstm outputs
```

```
    lstm_out = lstm_out.contiguous().view(-1, self.hidden_dim)
```

```
    # dropout and fully-connected layer
```

```
    out = self.dropout(lstm_out)
```

```
    out = self.fc(out)
```

```
    # sigmoid function
```

```
    sig_out = self.sig(out)
```

```
    # reshape to be batch_size first
```

```
    sig_out = sig_out.view(batch_size, -1)
```

```
    sig_out = sig_out[:, -1] # get last batch of labels
```

```
    # return last sigmoid output and hidden state
```

```
    return sig_out, hidden
```

### **forward explanation**

So, first, I'm getting the batch\_size of my input x, which I'll use for shaping my data. Then, I'm passing x through the embedding layer first, to get my embeddings as output

These embeddings are passed to my lstm layer, alongside a hidden state, and this returns an lstm\_output and a new hidden state! Then I'm going to stack up the outputs of my LSTM to pass to my last linear layer.

Then I keep going, passing the reshaped lstm\_output to a dropout layer and my linear layer, which should return a specified number of outputs that I will pass to my sigmoid activation function.

Now, I want to make sure that I'm returning *only* the **last** of these sigmoid outputs for a batch of input data, so, I'm going to shape these outputs into a shape that is batch\_size first. Then I'm getting the last batch by called `sig_out[:, -1]`, and that's going to give me the batch of last labels that I want!

Finally, I am returning that output and the hidden state produced by the LSTM layer.

### **init\_hidden**

That completes my forward function and then I have one more: `init_hidden` and this is just the same as you've seen before. The hidden and cell states of an LSTM are a tuple of values and each of these is size (n\_layers by batch\_size, by hidden\_dim). I'm initializing these hidden weights to all zeros, and moving to a gpu if available.

```
def init_hidden(self, batch_size):  
    """ Initializes hidden state """  
  
    # Create two new tensors with sizes n_layers x batch_size x hidden_dim,  
    # initialized to zero, for hidden state and cell state of LSTM  
    weight = next(self.parameters()).data  
  
    if (train_on_gpu):  
        hidden = (weight.new(self.n_layers, batch_size, self.hidden_dim).zero_().cuda(),  
                  weight.new(self.n_layers, batch_size, self.hidden_dim).zero_().cuda())  
    else:  
        hidden = (weight.new(self.n_layers, batch_size, self.hidden_dim).zero_(),  
                  weight.new(self.n_layers, batch_size, self.hidden_dim).zero_())  
  
    return hidden
```

After this, I'm ready to instantiate and train this model, you should see if you can decide on good hyperparameters of your own, and then check out the solution code, next!

## **Training the Model**

## Hyperparameters

After defining my model, next I should instantiate it with some hyperparameters.

*# Instantiate the model w/ hyperparams*

`vocab_size = len(vocab_to_int)+1 # +1 for the 0 padding + our word tokens`

`output_size = 1`

`embedding_dim = 400`

`hidden_dim = 256`

`n_layers = 2`

`net = SentimentRNN(vocab_size, output_size, embedding_dim, hidden_dim, n_layers)`

`print(net)`

This should look familiar, but the main thing to note here is our `vocab_size`.

This is actually the length of our `vocab_to_int` dictionary (all our unique words) **plus one** to account for the 0-token that we added, when we padded our input features. So, if you do data pre-processing, you may end up with one or two extra, special tokens that you'll need to account for, in this parameter!

Then, I want my `output_size` to be 1; this will be a sigmoid value between 0 and 1, indicating whether a review is positive or negative.

Then I have my embedding and hidden dimension. The embedding dimension is just a smaller representation of my vocabulary of 70k words and I think any value between like 200 and 500 or so would work, here. I've chosen 400. Similarly, for our hidden dimension, I think 256 hidden features should be enough to distinguish between positive and negative reviews.

I'm also choosing to make a 2 layer LSTM. Finally, I'm instantiating my model and printing it out to make sure everything looks good.

```
In [17]: # Instantiate the model w/ hyperparams
vocab_size = len(vocab_to_int)+1 # +1 for the 0 padding + our word tokens
output_size = 1
embedding_dim = 400
hidden_dim = 256
n_layers = 2

net = SentimentRNN(vocab_size, output_size, embedding_dim, hidden_dim, n_layers)
print(net)

SentimentRNN(
  (embedding): Embedding(74073, 400)
  (lstm): LSTM(400, 256, num_layers=2, batch_first=True, dropout=0.5)
  (fc): Linear(in_features=256, out_features=1, bias=True)
  (sig): Sigmoid()
)
```

## [Model hyperparameters](#)

### Training and Optimization

The training code, should look pretty familiar. One new detail is that, we'll be using a new kind of cross entropy loss that is designed to work with a single Sigmoid output.

[BCELoss](#), or **Binary Cross Entropy Loss**, applies cross entropy loss to a single value between 0 and 1.

We'll define an Adam optimizer, as usual.

*# loss and optimization functions*

`lr=0.001`

```
criterion = nn.BCELoss()
```

```
optimizer = torch.optim.Adam(net.parameters(), lr=lr)
```

### Output, target format

You should also notice that, in the training loop, we are making sure that our outputs are squeezed so that they do not have an empty dimension output.squeeze() and the labels are float tensors, labels.float(). Then we perform backpropagation as usual.

### Train and eval mode

Below, you can also see that we switch between train and evaluation mode when the model is training versus when it is being evaluated on validation data!

### Training Loop

Below, you'll see a usual training loop.

I'm actually only going to do four epochs of training because that's about when I noticed the validation loss stop decreasing.

- You can see that I am initializing my hidden state before entering the batch loop then have my usual detachment from history for the hidden state and backpropagation steps.
- I'm getting my input and label data from my train\_dataloader. Then applying my model to the inputs and comparing the outputs and the true labels.
- I also have some code that checks performance on my validation set, which, if you want, may be a great thing to use to decide when to stop training or which best model to save!

*# training params*

`epochs = 4 # 3-4 is approx where I noticed the validation loss stop decreasing`



```

counter = 0
print_every = 100
clip=5 # gradient clipping

# move model to GPU, if available
if(train_on_gpu):
    net.cuda()

net.train()
# train for some number of epochs
for e in range(epochs):
    # initialize hidden state
    h = net.init_hidden(batch_size)

    # batch loop
    for inputs, labels in train_loader:
        counter += 1

        if(train_on_gpu):
            inputs, labels = inputs.cuda(), labels.cuda()

        # Creating new variables for the hidden state, otherwise
        # we'd backprop through the entire training history
        h = tuple([each.data for each in h])

        # zero accumulated gradients
        net.zero_grad()

        # get the output from the model

```

```
output, h = net(inputs, h)
```

```
# calculate the loss and perform backprop
```

```
loss = criterion(output.squeeze(), labels.float())
```

```
loss.backward()
```

```
# `clip_grad_norm` helps prevent the exploding gradient problem in RNNs / LSTMs.
```

```
nn.utils.clip_grad_norm_(net.parameters(), clip)
```

```
optimizer.step()
```

```
# loss stats
```

```
if counter % print_every == 0:
```

```
    # Get validation loss
```

```
    val_h = net.init_hidden(batch_size)
```

```
    val_losses = []
```

```
    net.eval()
```

```
    for inputs, labels in valid_loader:
```

```
        # Creating new variables for the hidden state, otherwise
```

```
        # we'd backprop through the entire training history
```

```
        val_h = tuple([each.data for each in val_h])
```

```
    if(train_on_gpu):
```

```
        inputs, labels = inputs.cuda(), labels.cuda()
```

```
    output, val_h = net(inputs, val_h)
```

```
    val_loss = criterion(output.squeeze(), labels.float())
```

```
    val_losses.append(val_loss.item())
```

```
net.train()
```

```

print("Epoch: {}/{}...".format(e+1, epochs),
      "Step: {}...".format(counter),
      "Loss: {:.6f}...".format(loss.item()),
      "Val Loss: {:.6f}".format(np.mean(val_losses)))

```

Make sure to take a look at how training **and** validation loss decrease during training! Then, once you're satisfied with your trained model, you can test it out in a couple ways to see how it behaves on new data!

## Consult the Solution Code

To take a closer look at this solution, feel free to check out the solution workspace or click [here](#) to see it as a webpage.

## Testing the Trained Model

I want to show you two great ways to test: using test data and using inference. The first is similar to what you've seen in our CNN lessons. I am iterating through the test data in the `test_loader`, recording the test loss and calculating the accuracy based on how many labels this model got correct!

I'm doing this by looking at the **rounded value** of our output. Recall that this is a sigmoid output between 0-1 and so rounding this value will give us an integer that is the most likely label: 0 or 1. Then I'm comparing that predicted label to the true label; if it matches, I record that as a correctly-labeled test review.

```

# Get test data loss and accuracy

test_losses = [] # track loss
num_correct = 0

# init hidden state
h = net.init_hidden(batch_size)

net.eval()
# iterate over test data
for inputs, labels in test_loader:

    # Creating new variables for the hidden state, otherwise
    # we'd backprop through the entire training history
    h = tuple([each.data for each in h])

    if(train_on_gpu):
        inputs, labels = inputs.cuda(), labels.cuda()

    # get predicted outputs
    output, h = net(inputs, h)

    # calculate loss
    test_loss = criterion(output.squeeze(), labels.float())
    test_losses.append(test_loss.item())

```

```

# convert output probabilities to predicted class (0 or 1)
pred = torch.round(output.squeeze()) # rounds to the nearest integer

# compare predictions to true label
correct_tensor = pred.eq(labels.float().view_as(pred))
correct = np.squeeze(correct_tensor.numpy()) if not train_on_gpu else np.squeeze(correct_tensor.cpu().numpy())
num_correct += np.sum(correct)

# -- stats! -- ##
# avg test loss
print("Test loss: {:.3f}".format(np.mean(test_losses)))

# accuracy over all test data
test_acc = num_correct/len(test_loader.dataset)
print("Test accuracy: {:.3f}".format(test_acc))

```

Below, I'm printing out the average test loss and the accuracy, which is just the number of correctly classified items divided by the number of pieces of test data, total.

We can see that the test loss is `0.516` and the accuracy is about **81.1%** !

```

# -- stats! -- ##
# avg test loss
print("Test loss: {:.3f}".format(np.mean(test_losses)))

# accuracy over all test data
test_acc = num_correct/len(test_loader.dataset)
print("Test accuracy: {:.3f}".format(test_acc))

```

```

Test loss: 0.516
Test accuracy: 0.811

```

### Test results

Next, you're ready for your last task! Which is to define a `predict` function to perform inference on any given text review!

### Exercise: Inference on a test review

You can change this `test_review` to any text that you want. Read it and think: is it pos or neg? Then see if your model predicts correctly!

**Exercise:** Write a `predict` function that takes in a trained net, a plain `text_review`, and a sequence length, and prints out a custom statement for a positive or negative review!

- You can use any functions that you've already defined or define any helper functions you want to complete `predict`, but it should just take in a trained net, a text review, and a sequence length.

```
def predict(net, test_review, sequence_length=200):
    """ Prints out whether a give review is predicted to be
        positive or negative in sentiment, using a trained model.

        params:
        net - A trained net
        test_review - a review made of normal text and punctuation
        sequence_length - the padded length of a review
    """

    # print custom response based on whether test_review is pos/neg
```

Try to solve this task on your own, then check out the solution, next!

## CLOUD COMPUTING:

### Login to the Instance

After launch, your instance may take a few minutes to initialize.

Once you see “2/2 checks passed” on the EC2 Management Console, your instance is ready for you to log in.

Status Checks ▾	Alarm Status	Public DNS (IPv4) ▾	IPv4 Public IP ▾
 2/2 checks ...	None	 ec2-52-8-198-63.us-we...	52.8.198.638

Instance Status Check and Public IP

Note the "IPv4 Public IP" address (in the format of “X.X.X.X”) on the EC2 Dashboard.

From a terminal, navigate to the location where you stored your .pem file. (For example, if you put your .pem file on your Desktop, `cd ~/Desktop/` will move you to the correct directory.)

Type `ssh -i YourKeyName.pem ubuntu@X.X.X.X`, where:

- `X.X.X.X` is the IPv4 Public IP found in AWS, and
- `YourKeyName.pem` is the name of your .pem file.

Note that if you've used a different AMI or specified a username, `ubuntu` will be replaced with the username, such as `ec2-user` for some Amazon AMI's. You would then instead enter `ssh -i YourKeyName.pem ec2-user@X.X.X.X`

## Configure Jupyter notebook settings

In your instance, in order to create a config file for your Jupyter notebook settings, type: `jupyter notebook --generate-config`.

Then, to change the IP address config setting for notebooks (this is just a fancy one-line command to perform an exact string match replacement; you could do the same thing manually using vi/vim/nano/etc.), type: `sed -ie "s/#c.NotebookApp.ip = 'localhost'/#c.NotebookApp.ip = '*'/'g" ~/.jupyter/jupyter_notebook_config.py`

## Test the Instance

Make sure everything is working properly by verifying that the instance can run a notebook.

### On the EC2 instance

- Clone a GitHub repository
- `git clone https://github.com/udacity/deep-learning-v2-pytorch.git`
- Enter the repo directory, and the CNN subdirectory
- `cd deep-learning-pytorch`
- `cd cnn-content`
- Install the requirements
- `sudo python3 -m pip install -r requirements.txt`
- Start Jupyter notebook
- `jupyter notebook --ip=0.0.0.0 --no-browser`

### From your local machine

- You will need the token generated by your jupyter notebook to access it. On your instance terminal, there will be the following line: `Copy/paste this URL into your browser when you connect for the first time, to login with a token:`. Copy everything starting with the `:8888/?token=`.
- Access the Jupyter notebook index from your web browser by visiting: `X.X.X.X:8888/?token=...` (where X.X.X.X is the IP address of your EC2 instance and everything starting with `:8888/?token=` is what you just copied).
- Click on a folder, like "mnist", to enter it and select a notebook, such as the "mnist\_mlp.ipynb" notebook.
- Run each cell in the notebook.  
For some notebooks, you should see a marked decrease in training time when compared to running the same cells using a typical CPU!

**NOTE:** Windows users may prefer connecting via the GUI utility PuTTY, by following [these instructions](#).

---

### Important: Cost

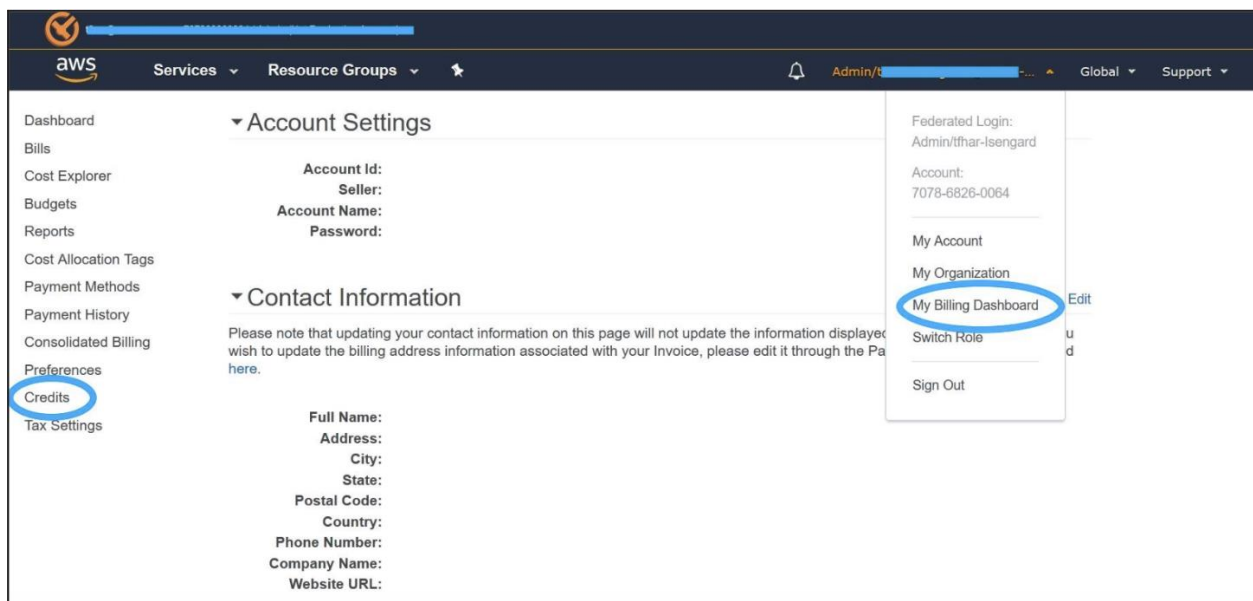
From this point on, AWS will charge you for a running an EC2 instance. You can find the details on the [EC2 On-Demand Pricing page](#).

Most importantly, remember to **stop** (i.e. shutdown) your instances when you are not using them. Otherwise, your instances might run for a day or a week or a month without you remembering, and you'll wind up with a large bill!

AWS charges primarily for running instances, so most of the charges will cease once you stop the instance. However, there are smaller storage charges that continue to accrue until you "terminate" (i.e. delete) the instance.

## Create an AWS Account

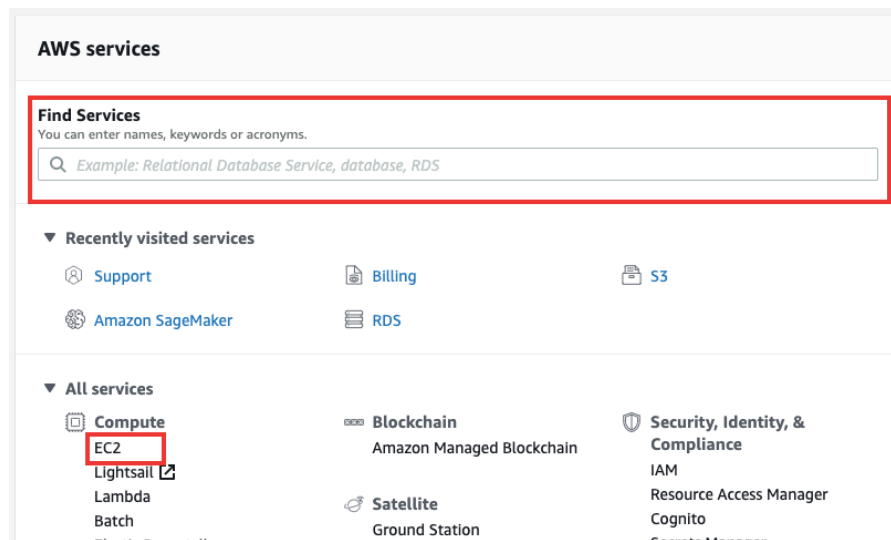
1. Open a regular AWS account (if you don't already have one) following the instructions via the [Amazon Web Service Help Center](#)
    2. You will need a promo code from us so you can apply it to your account. To request a promo code, you can submit a support ticket [here](#).
  - Under the "**Reason for Contact**" field, choose "**Other**", then choose "**External Tools**" in the dropdown.
  - When the "**External Tools**" field appears, select "**AWS**".
  - Please note that a regular AWS account will receive a promo code from Udacity with a fixed amount of AWS credits.
3. To apply your promo code, follow below:
    - Click "**Credits**" on the left side of the screen and enter the promo-code you received, then hit "**redeem**".
    - Refresh the page and you will be able to view your credits under: Below are all the credits you have redeemed with AWS. Credits will automatically be applied to your bill.



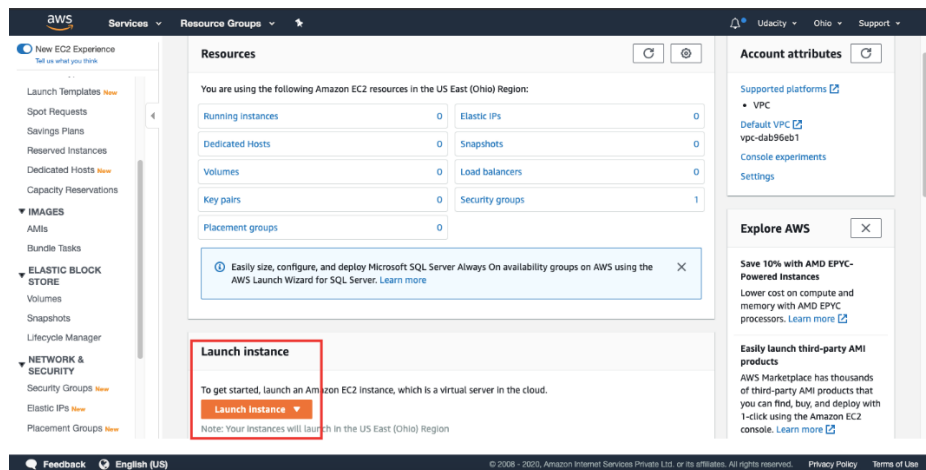
Amazon Web Services (AWS) is an Infrastructure as a Service (IaaS) and Platform as a Service (PaaS) provider. At this point, we may think of a "cloud" as a geographically distributed set of data centers that host different Virtual Machines (VMs) from different users. AWS has divided the entire world into several geographical regions. Each region has many availability zones, which in turn, comprises few data centers. Each data center has hundreds of servers, each of which hosts thousands of VMs dynamically.

## A. What is EC2?

AWS Elastic Compute Cloud (EC2) is a hosted service that allows you to launch **Virtual Machines (or an "instances")**, including instances with attached GPUs. You can get started by logging into the [AWS Console](#) and search for the "EC2" service, as shown below.



AWS Console - Select an AWS Service



AWS Console - EC2 Dashboard



## A.1. Launch an EC2 Instance

You can launch an instance in 7 simple steps, as follows:

1. Choose an Amazon Machine Image (AMI) - An AMI is a template that contains the software configuration (operating system, application server, and applications) required to launch your instance.
2. Choose an Instance Type - Instance Type offers varying combinations of CPUs, memory (GB), storage (GB), types of network performance, and availability of IPv6 support. AWS offers a variety of Instance Types, broadly categorized in 5 categories. You can choose an Instance Type that fits our use case. The specific type of GPU instance you should launch for this tutorial is called “**p2.xlarge**”.
3. Configure Instance Details - Provide the instance count and configuration details, such as, network, subnet, behavior, monitoring, etc.
4. Add Storage - You can choose to attach either *SSD* or *Standard Magnetic* drive to your instance.
5. Add Tags - A tag serves as a *label* that you can attach to multiple AWS resources, such as volumes, instances or both.
6. Configure Security Group - Attach a set of firewall rules to your instance(s) that controls the incoming traffic to your instance(s).
7. Review - Review your instance launch details before the launch.

## A.2. What is an Amazon Machine Image?

**Amazon Machine Image** or **AMI** is a template for an operating system and basic services (e.g., an application server and specific applications). By running an AMI instance, the AMI will be running "as a virtual server in the cloud" (as per [AWS Documentation](#)).

We will use this [Deep Learning AMI \(Amazon Linux\)](#) to define the operating system for your instance and to make use of its pre-installed software. In order to use this AMI, you must change your AWS region to one of the following (and you are encouraged to select the region in the list that is closest to you):

- EU (Ireland)
- Asia Pacific (Seoul)
- Asia Pacific (Tokyo)
- Asia Pacific (Sydney)
- US East (N. Virginia)
- US East (Ohio)
- US West (Oregon)

If you are unsure, please check the [AWS documentation](#) to confirm which region may be closest to you.

After changing your AWS region, view your **EC2 Service Limit report** at [this link](#), and find your "Current Limit" for the p2.xlarge instance type.

## A.3. Instance Type - What is a P2 instance?

P2 are powerful and scalable parallel processing GPU instances. You can read more about these on [AWS Documentation](#).

---

## B. Shut Down EC2 Instances, if not in use

**Note:** We recommend you shut down every resource (e.g., EC2 instances, or any other hosted service) on the AWS cloud immediately after the usage, otherwise you will run out of your free promo credits.

Even if you are in the middle of the project and need to step away, **PLEASE SHUT DOWN YOUR EC2 INSTANCE**. You can re-instantiate later. We have provided adequate credits to allow you to complete your projects.

## C. AWS Service Utilization Limits

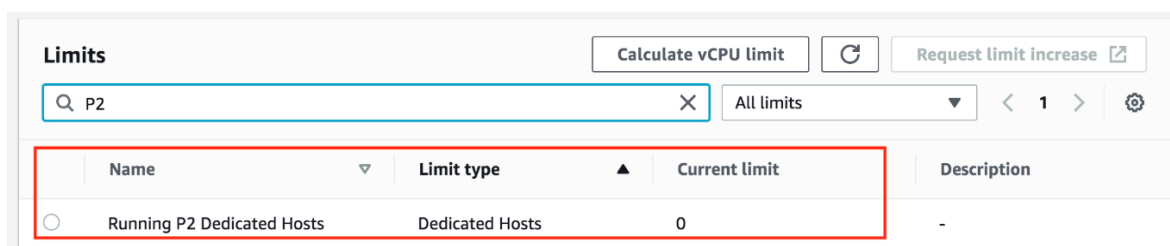
You need to understand the way AWS imposes *utilization quotas* (limits) on almost all of its services.

- AWS provides default quotas, formerly referred to as *limits*, for each AWS service.
- Importantly, **each quota is region-specific**.
- There are three ways to check your quotas, as mentioned [here](#), i). Service Endpoints and Quotas, ii). Service Quotas console, and iii). AWS CLI commands.
- We recommend you to check the quotas for AWS EC2 at [Service Endpoints and Quotas](#) page.
- Alternatively, for EC2 service, you can visit [Amazon EC2 Service Limits](#) that describes how to view the current quota (limit) or request an increase in quota.

### C.1. Increase EC2 Instance Limits

#### C.1.1. View Your Current Limit

After changing your AWS region, view your [EC2 Service Limit report at this link](#), and find your "Current Limit" for the p2.xlarge instance type. By default, **AWS sets a limit of 0 on the number of p2.xlarge instances a user can run**, which effectively prevents you from launching this instance.



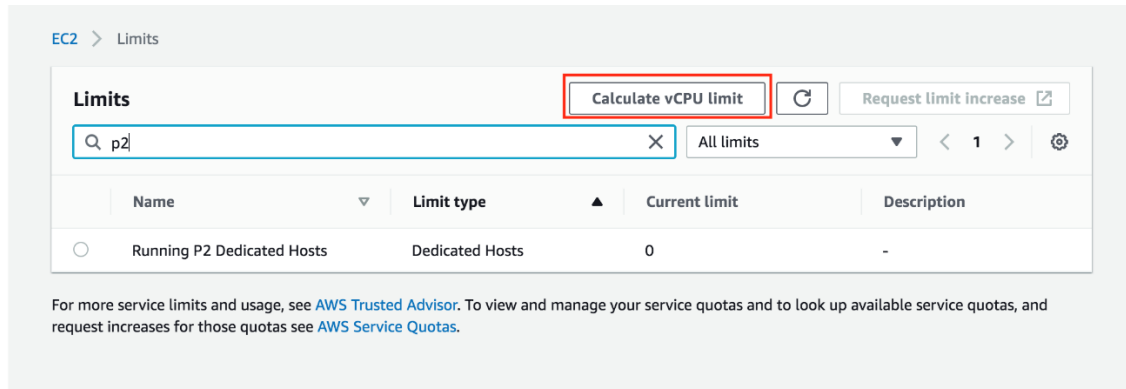
The screenshot shows the AWS Service Limits console. At the top, there's a search bar with 'P2' entered. Below the search bar, there's a table with columns: Name, Limit type, Current limit, and Description. The first row in the table is 'Running P2 Dedicated Hosts' with a limit type of 'Dedicated Hosts' and a current limit of '0'. A red box highlights the first row of the table.

Name	Limit type	Current limit	Description
Running P2 Dedicated Hosts	Dedicated Hosts	0	-

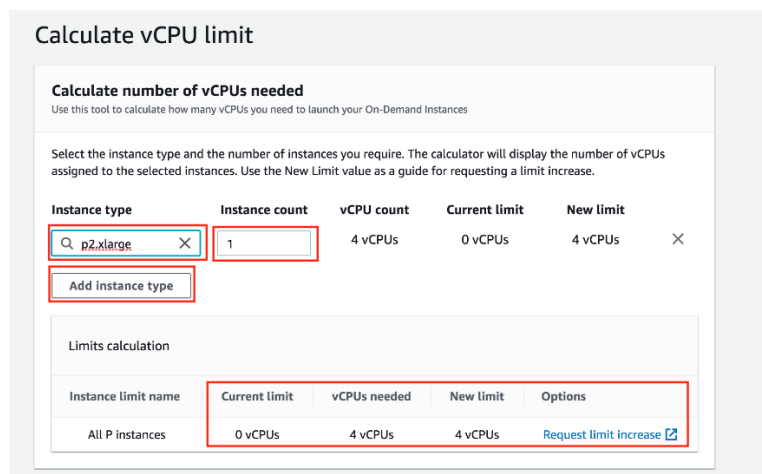
#### C.1.2. Submit a Limit Increase Request

If your limit of p2.xlarge instances is 0, you'll need to increase the limit before you can launch an instance.

1. From the EC2 Service Limits page, **CLICK TO EDIT Calculate vCPU limit** on top.



2. The following screen pops up. In the **Instance type** search for p2.xlarge and the **Instance count** to 1, verify the **New limit** is showing **4 vCPUs** and then click on **Request limit increase** *Note: You won't be charged for requesting a limit increase. AWS will only charge you once you have launched the instance.*



3. This will open the following screen. Click on the **Service limit increase** if it is not already selected.

**Create case** [Info](#)

Account and billing support

Assistance with account and billing-related enquiries

**Service limit increase**

Requests to increase the service limit of your AWS resources

Technical support

Service-related technical issues and third-party applications

Unavailable under the Basic Support Plan

**Case classification**

Limit type

EC2 Instances

Severity [Info](#)

The severity levels available are determined by your support subscription.

General question

**Useful links**

- [Amazon EC2 Service Limits](#)
- [Amazon EC2 On-Demand Instance limits](#)
- [vCPU-based On-Demand Instance Limits FAQ](#)

**Requests**

4. In **Requests** section pick your "Primary Instance type" as "All P instances" and "1" for "New limit value" **Note:** *If you have never launched an instance of any type on AWS, you might receive an email from AWS Support asking you to initialize your account by creating an instance before they approve the limit increase.*

**Requests**

To request additional limit increases for the same limit type, choose **Add another request**. To request an increase for a different limit type, create a separate limit increase request.

**Request 1** [Remove](#)

Region

US East (Ohio)

Primary Instance Type

All P instances

Limit

Instance Limit

New limit value

1

[Add another request](#)

### C.1.3. Wait for the Approval

You must wait for AWS to approve your Limit Increase Request. AWS typically approves these requests within 48 hours.

IMPORTANT NOTICE: This is the current AWS UI as of April 6th, 2020. The AWS UI is subject to change on a regular basis. We advise students to refer to AWS documentation for the above process.

## Cloud Providers, AWS

If you're enrolled in the complete program, you can use your AWS Promotional Credits to get started with AWS Machine Learning services such as Amazon SageMaker. Amazon SageMaker is a fully-managed machine learning platform that enables developers and data scientists to quickly and easily build, train, and deploy machine learning models at any scale. It removes all the barriers that typically slow down developers who want to use machine learning, so it's a perfect starting point on your machine learning journey.

[Click here](#) to learn more!



---

[The cloud can connect data storage, analysis, and trained deep learning models.](#)

### Deployment via the Cloud

Using a cloud provider like AWS to spin up a GPU instance is really useful for training your own neural networks on the latest GPU hardware. There is one other benefit to using a cloud provider and that's **deployment**. Often, you'll want to deploy your pre-trained models so that they can respond to user data on a website! For example, [this demo](#) that uses a trained RNN to generate new sketches given user input.

You'll learn all about deployment later in this program, so we'll revisit the necessary instance-creation information, when we get there. The goal of this short, text lesson is really about getting you acquainted with GPU instances (should you want to use them), and introducing you to cloud providers like AWS.

## Launch an Instance

Once AWS approves your GPU Limit Increase Request, you can start the process of launching your instance.

Visit the [EC2 Management Console](#), and click on the “Launch Instance” button.

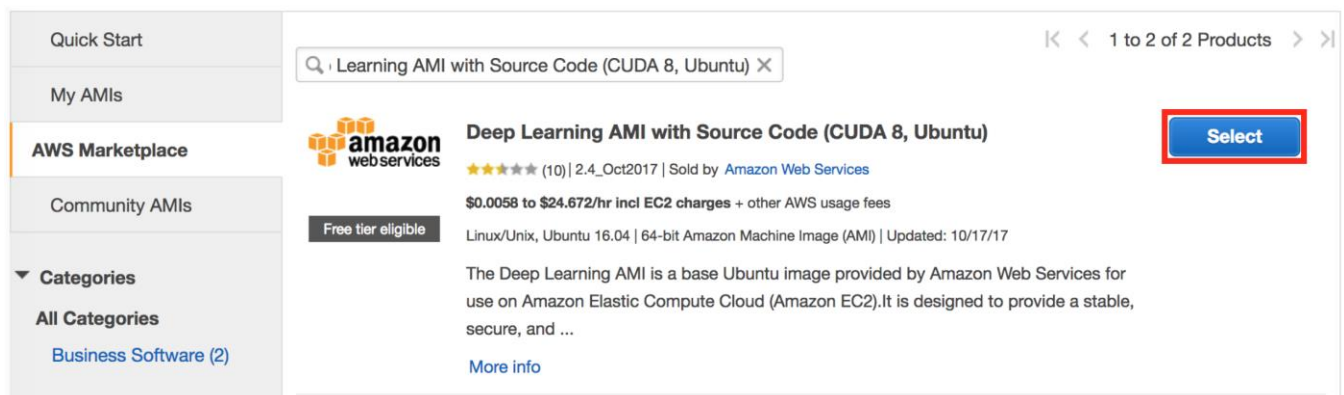
## Create Instance

To start using Amazon EC2 you will want to launch a virtual server, known as an Amazon EC2 instance.



Next, you must choose an AMI (Amazon Machine Image) which defines the operating system for your instance, as well as any configurations and pre-installed software.

Click on **AWS Marketplace**, and search for **Deep Learning AMI with Source Code (CUDA 8, Ubuntu)**. Once you find the appropriate AMI, click on the "Select" button.



This [Amazon Machine Image \(AMI\)](#) contains all the environment files and drivers for you to train on a GPU. It has [cuDNN](#), and many the other packages required for this course. Any additional packages required for specific projects will be detailed in the appropriate project instructions.

## Select the Instance Type

You must next choose an instance type, which is the hardware on which the AMI will run.

Filter the instance list to only show “GPU compute”:

Filter by:

GPU compute

Current generation

Currently selected: p2.xlarge (11.75 ECUs, 4 vCPUs, 2.7 GHz,

	Family	Type	
<input checked="" type="checkbox"/>	GPU compute	p2.xlarge	
<input type="checkbox"/>	GPU compute	p2.8xlarge	
<input type="checkbox"/>	GPU compute	p2.16xlarge	

Select the p2.xlarge instance type:

Filter by:

GPU compute

Current generation

Show/Hide Columns

Currently selected: p2.xlarge (11.75 ECUs, 4 vCPUs, 2.7 GHz, E5-2686v4, 61 GiB memory, EBS only)

Note: The vendor recommends using a **p2.xlarge** instance (or larger) for the best experience with this p

	Family	Type	vCPUs	Memory (GiB)	Instance Storage (GB)	E
<input checked="" type="checkbox"/>	GPU compute	p2.xlarge	4	61	EBS only	
<input type="checkbox"/>	GPU compute	p2.8xlarge	32	488	EBS only	
<input type="checkbox"/>	GPU compute	p2.16xlarge	64	732	EBS only	

Finally, click on the "Review and Launch" button:

[Cancel](#)[Previous](#)[Review and Launch](#)[Next: Configure Instance Details](#)

## Configure the Security Group

Running and accessing a Jupyter notebook from AWS requires special configurations.

By default, AWS restricts access to most ports on an EC2 instance. In order to access the Jupyter notebook, you must configure the AWS Security Group to allow access to port 8888.

Click on "Edit security groups".

### ► Security Groups

On the "Configure Security Group" page:

1. Select "Create a **new** security group"
2. Set the "Security group name" (i.e. "Jupyter")
3. Click "Add Rule"
4. Set a "Custom TCP Rule"
1. Set the "Port Range" to "8888"
2. Select "Anywhere" as the "Source"
5. Click "Review and Launch" (again)

Assign a security group: ☒ Create a new security group

☐ Select an existing security group

Security group name:

Description:

Type <small>i</small>	Protocol <small>i</small>	Port Range <small>i</small>	Source <small>i</small>	
SSH	TCP	22	Custom 0.0.0.0/0	✕
Custom TCP Rule	TCP	8888	Anywhere 0.0.0.0/0	✕

Add Rule

## Launch the Instance

Click on the "Launch" button to launch your GPU instance.



Cancel

Previous

Launch

## Create an Authentication Key Pair

AWS will ask if you'd like to specify an authentication key pair. You'll need to do so in order to access your instance, so you select "Create a new key pair" and click the "Download Key Pair" button. This will download a .pem file, which you'll need to be able to access your instance.

Download Key Pair

Your key name

Key pair name

Create a new key pair



about removing existing key pairs from a public AMI.

Note: The selected key pair will be added to the set of keys authorized for this instance. Learn more

Move the .pem file to a secure and easily remembered location on your computer; you'll need to access your instance through the location you select.

After the .pem file has been downloaded, click the "Launch Instances" button.

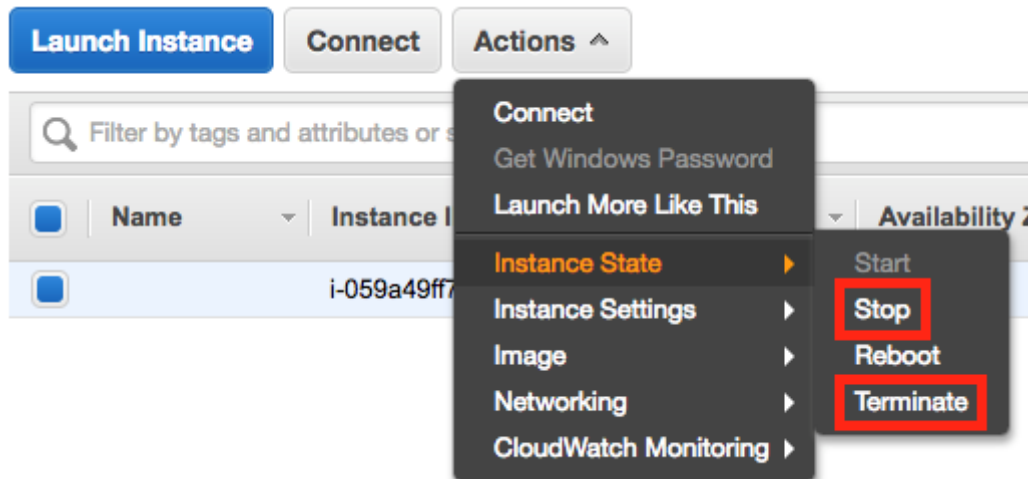
Click the "View Instances" button to go to the EC2 Management Console and watch your instance boot.

## Be Careful!

From this point on, AWS will charge you for running this EC2 instance. You can find the details on the [EC2 On-Demand Pricing page](#).

Most importantly, remember to "stop" (i.e. shutdown) your instances when you are not using them. Otherwise, your instances might run for a day, week, month, or longer without you remembering, and you'll wind up with a large bill!

AWS charges primarily for running instances, so most of the charges will cease once you stop the instance. However, there are smaller storage charges that continue to accrue until you "terminate" (i.e. delete) the instance.



There is no way to limit AWS to only a certain budget and have it auto-shutdown when it hits that threshold. However, you can set [AWS Billing Alarms](#).

### Login to the Instance

After launch, your instance may take a few minutes to initialize.

Once you see “2/2 checks passed” on the EC2 Management Console, your instance is ready for you to log in.

Status Checks	Alarm Status	Public DNS (IPv4)	IPv4 Public IP
✓ 2/2 checks ...	None	ec2-52-8-198-63.us-we...	52.8.198.638

### Instance Status Check and Public IP

Note the "IPv4 Public IP" address (in the format of “X.X.X.X”) on the EC2 Dashboard.

From a terminal, navigate to the location where you stored your .pem file. (For example, if you put your .pem file on your Desktop, `cd ~/Desktop/` will move you to the correct directory.)

Type `ssh -i YourKeyName.pem ubuntu@X.X.X.X`, where:

- X.X.X.X is the IPv4 Public IP found in AWS, and
- YourKeyName.pem is the name of your .pem file.

Note that if you've used a different AMI or specified a username, ubuntu will be replaced with the username, such as ec2-user for some Amazon AMI's. You would then instead enter `ssh -i YourKeyName.pem ec2-user@X.X.X.X`

## Configure Jupyter notebook settings

In your instance, in order to create a config file for your Jupyter notebook settings, type: `jupyter notebook --generate-config`.

Then, to change the IP address config setting for notebooks (this is just a fancy one-line command to perform an exact string match replacement; you could do the same thing manually using vi/vim/nano/etc.), type: `sed -ie "s/#c.NotebookApp.ip = 'localhost'/#c.NotebookApp.ip = '*' /g" ~/.jupyter/jupyter_notebook_config.py`

## Test the Instance

Make sure everything is working properly by verifying that the instance can run a notebook.

### On the EC2 instance

- Clone a GitHub repository
  - `git clone https://github.com/udacity/deep-learning-v2-pytorch.git`
- Enter the repo directory, and the CNN subdirectory
  - `cd deep-learning-pytorch`
  - `cd cnn-content`
- Install the requirements
  - `sudo python3 -m pip install -r requirements.txt`
- Start Jupyter notebook
  - `jupyter notebook --ip=0.0.0.0 --no-browser`

### From your local machine

- You will need the token generated by your jupyter notebook to access it. On your instance terminal, there will be the following line: Copy/paste this URL into your browser when you connect for the first time, to login with a token:. Copy everything starting with the `:8888/?token=`.
- Access the Jupyter notebook index from your web browser by visiting: `X.X.X.X:8888/?token=...` (where X.X.X.X is the IP address of your EC2 instance and everything starting with `:8888/?token=` is what you just copied).
- Click on a folder, like "mnist", to enter it and select a notebook, such as the "mnist\_mlp.ipynb" notebook.
- Run each cell in the notebook.

For some notebooks, you should see a marked decrease in training time when compared to running the same cells using a typical CPU!

**NOTE:** Windows users may prefer connecting via the GUI utility PuTTY, by following [these instructions](#).

---

### **Important: Cost**

**From this point on, AWS will charge you for a running an EC2 instance.** You can find the details on the [EC2 On-Demand Pricing page](#).

Most importantly, remember to stop (i.e. shutdown) your instances when you are not using them. Otherwise, your instances might run for a day or a week or a month without you remembering, and you'll wind up with a large bill!

AWS charges primarily for running instances, so most of the charges will cease once you stop the instance. However, there are smaller storage charges that continue to accrue until you "terminate" (i.e. delete) the instance.