[nc-state logo](https://www.ncsu.edu/)

Deep Learning

[*Christopher G. Healey*](http://www.csc.ncsu.edu/faculty/healey)

* **Introduction**
* **Setup**
* **DNNs**
* **Backpropegation**
* **PyTorch**
* **Wheat Ex**
* **Images Ex**
* **CIFAR FCN Ex**
* **CIFAR CNN Ex**
* **RNNs/LSTMs**

**Introduction**

This module will introduce you to deep learning and deep neural networks using [PyTorch](https://pytorch.org/), a Python-based open source deep learning package created by Facebook. We will use PyTorch to introduce both fully connected deep neural networks (FCNs) and convolutional deep neural networks (CNNs) for image classification tasks.

**Setup**

Before you can use PyTorch, you must download and install its relevant modules in Anaconda. To do this, open an Anaconda command prompt and issue the command

conda install pytorch torchvision -c pytorch

You can review the [instructions on the PyTorch web site](https://pytorch.org/get-started/locally/#anaconda-1) for more information, as well as commands you can issue to verify the install has been performed correctly.

**Deep Neural Networks**

Deep neural networks (DNNs) are a relatively new area of focused research, based off the foundations of artificial neural networks (ANNs). ANNs and DNNs fall within the broad area of supervised machine learning algorithms. The original idea for a *neural network* was [proposed by McCulloch and Pitts in 1943](https://www.cs.cmu.edu/~./epxing/Class/10715/reading/McCulloch.and.Pitts.pdf), based off a biological estimate of how neurons in the brain were hypothesized to function. The problem with the McCulloch-Pitts model was its inability to easily learn. To address this, [Rosenblatt proposed the Perceptron](http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.335.3398&rep=rep1&type=pdf), adding weights that allowed the ANN to "learn" by increasing and decreasing weights based on how well the current network categorized relative to a known, correct label (*i.e.*, supervised learning).

In 1959, Widrow and Hoff developed MADALINE at Stanford University to remove echoes on phone lines. Interestingly, MADALINE is still in use. Careful analysis of MADALINE showed that it found a set of weights for a number of inputs, which is analogous to linear regression. In other words, ANNs at this point could not solve more complex, non-linear problems. [Marvin Minsky and Seymour Papert at MIT](https://mitpress.mit.edu/books/perceptrons) proved this theoretically, effectively silencing research on neural networks for many years.

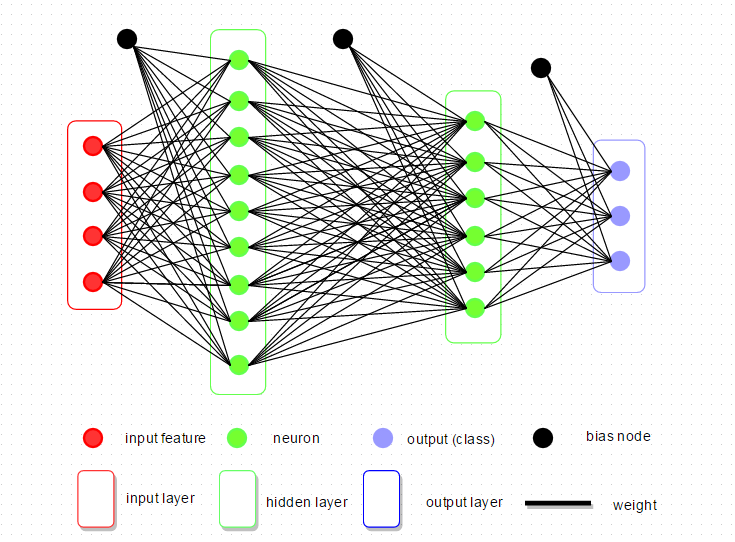
Moving into the 1980s, multilayer neural networks with *hidden layers* were developed. This was critical, since it provides one of the key advantages of a DNN: the ability to automatically create relevant features from the initial inputs. This is also where the terminology "deep" comes from. The intuition is that each hidden layer in a DNN uses results from the previous layer, allowing it to start with highly detailed features, then use those to proceed to identify more abstract elements. This led to another problem, however. Although it was understood how to train single-layer ANNs, it was not know how to adjust weights, biases, and activations on a multi-layer DNN.

To address this, the idea of backpropagation, which distributes error throughout the network, was [proposed by Rumelhart, Hinton, and Williams](https://web.stanford.edu/class/psych209a/ReadingsByDate/02_06/PDPVolIChapter8.pdf) in 1986. In simple terms, we are using calculus to assign some of the blame for error in the output layer to each neuron in the previous hidden layer, further propagating error at that hidden layer to its parent and so on. Initially, stochastic gradient descent was used to find an optimal set of weights to minimize error, although other approaches have now been proposed. Backpropagation is performed at the possible cost of significantly increased training times. Because of this, we now use clusters of graphics processing units (GPUs) which support 1000s of parallel operations to train a DNN.

DNNs have been used to solve previously intractable problems in a wide range of areas, including:

* image recognition,
* speech recognition,
* game playing,
* control systems,
* driver-less cars, and
* unmanned aerial systems

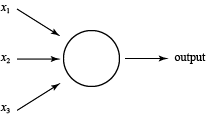
The basic structure of a DNN is made up of: an input layer; two or more hidden layers of neurons of some size; an output layer; edges connecting neurons between adjacent layers; activation values and biases at each neuron; weights on each edge.



This example shows a FCN with two hidden layers. More complicated DNNs like convolutional neural networks precede the FCN with a set of convolution operations to extract *features* which are selected, filtered, and then used as an input layer into a final FCN for classification. We will begin our exploration of DNNs with a PyTorch example that classifies images of handwritten numbers from 0 to 9 using a simple FCN.

***Neurons***

Neural networks, including deep neural networks, are made up of specialized perceptrons called neurons. A perceptron is an element that receives binary input(s) {x1,…,xn} and generates a binary output.



Weights {w1,…,wn} are normally applied to the inputs to define their *importance*. If the sum of the inputs multiplied by their weights exceeds a threshold or *bias* bi, the perceptron fires 1, otherwise it fires 0.

output={0if∑iwixi≤bi1if∑iwixi>bi

We will normally simplify this notation by moving the bias inside the equation and using a dot product for the sum.

output={0ifw⋅x−b≤01otherwise

Bias bi is a measure of how easy or difficult it is to make a perceptron fire. The larger bi, the more input or *activation* perceptron *i* needs to fire.

Although useful, perceptrons are inconvenient since both their inputs and outputs are binary. To address this, we normally convert perceptrons to neurons. Neurons accept continuous inputs and generate continuous outputs. To do this, neurons apply a function like sigmoid to produce smooth output over a fixed range. For example, "sigmoid" neurons accept continuous input and apply σ to the weighted sum of activation plus bias to produce continuous output on the range 0,…,1.

z=w⋅x+boutput=σ(z)=11+e−z=11+exp(−(w⋅x+b))

Intuitively, if z=w⋅x+b is large, e−z≈0 so σ(z)≈1 and if z is small σ(z)≈0, just like a perceptron. The difference is that σ is smooth over the range 0,…,1 so a small change in weights or biases Δwi or Δbi will produce a small change in output, versus a perceptron's potential to flip its output from 0 to 1 or vice versa.

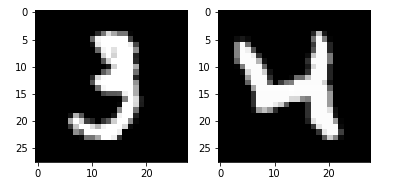
Δoutput≈∑i∂output∂wiΔwi+∂output∂biΔbi

That is, Δoutput is linear in Δw and Δb.

***Handwritten Number Recognition***

PyTorch's dataset repository includes the [MNIST database of handwritten images](http://yann.lecun.com/exdb/mnist/), which includes 60,000 training examples and 10,000 test examples. It is a very common dataset to use to test learning or pattern recognition algorithms on real-world data, without the need to hand-label a large training dataset.

Normally, images are processed with CNNs. However, we will use a simple FCN to recognize the handwritten images. This is done by converting each handwritten digit image into a set of pixel values, then converting that into a one-dimensional vector. This 1D vector acts as input to one or more hidden layers, filters like ReLU (rectified linear unit), and finally to an output layer with ten possible classifications representing the ten digits 0–9. Here are two simple examples of the digits in the MNIST dataset.



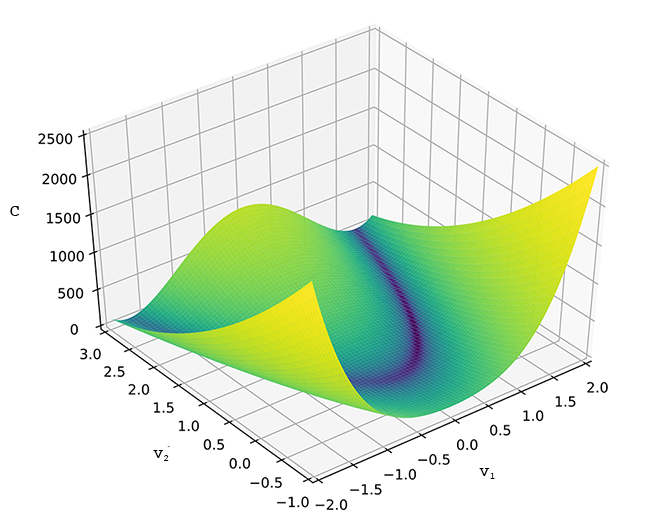
In our example neural network to "learn" MNIST images we have a 28×28=784-element input vector of greyscale intensities, an n=15 neuron hidden layer, and 10 output values representing a decision on which digit the input image represents. To train and validate the DNN model, MNIST provides 60,000 labeled training images and 10,000 labeled test images, denoted as x and y, respectively. Any individual xi is a 784-length vector and the corresponding yi is a 10-length output vector with all 0s except for the target digit position, which is 1.

Our goal is to train a neural network so output approximates yi=y(xi)∀xi in the training set. To do this, we define a cost function to measure accuracy or error for any given prediction.

C(w,b)=12n∑x||y(x)−a||2

where w, b, y, and a are weights, biases, known output vectors (labels), and predicted output vectors (activation) for all training input samples x. You should recognize this as the simple quadratic function mean squared error (MSE), where C(w,b)→0 as we improve our ability to predict correct output.

Obviously, when we start we expect C(w,b) to be large. To reduce it, we *optimize* w and b throughout the network using gradient descent. Ignoring for now the specifics of our neural net, how can we minimize C(v) where v are tunable parameters? For simplicity, assume v=(v1,v2) is a two-dimensional parameter vector. Plotting C(v) produces a valley-like image.



From any point on the valley, we want to move in a direction with the steepest slope towards the valley bottom (minimum C value). What happens when we move a small amount Δv1 in the v1 direction? Or a small amount Δv2 in the v2 direction?

ΔC≈∂C∂v1Δv1+∂C∂v2Δv2

We want C to be negative. Defining Δv=(Δv1,Δv2)T and gradient of descent ∇C=(∂C∂v1,∂C∂v2), we have

ΔC=∇C⋅Δv

Now, if we pick ϵ s.t.

Δv=−ϵ∇C

where ϵ is the *learning rate* and therefore Δv is a small movement in the direction of steepest gradient, then

ΔC=∇C⋅−ϵ∇C=−ϵ||∇C||2

Since ||∇C||2 is always positive, ΔC is always negative. Setting v→v′=v−ϵ∇C over and over, we will converge on a minimum C, as long as ϵ is not too big, causing us to "jump" back and forth over the minimum, or too small, causing convergence to be too expensive. Moreover, if we expand v into a vector of m>2 variables v=(v1,…,vm), this approach continuous to work.

ΔC=∇C⋅Δv∇C=(∂C∂v1,…,∂C∂vm)Δv=−ϵ∇Cv→v′=v−ϵ∇C

How do we extend this general gradient descent approach to our specific goal of optimizing C in a neural network. We simply rewrite the above equations in terms of w and b.

wi→wi′=wi−ϵ∂C∂wibi→bi′=bi−ϵ∂C∂bi

We can now walk backwards through the layers in the neural network, adjusting (backpropegating) weights and biases for each neuron. To do this, recall C=1n∑xCx and Cx=||yi(xi)−ai||22. To compute ∇C we compute ΔCx∀x and average the result.

∇C=1n∑x∇Cx

If the number of neurons n is very large, we may want to sample ∇Cx. This is called stochastic gradient descent, where we randomly choose m≪n training inputs to form a mini-batch (x1,…,xm), assuming

∑i=1m∇Cxim≈∑x∇Cxn=∇C∇C≈1m∑i=1m∇Cxiwi→wi′=wi=ϵm∑i∂Cxi∂wibi→bi′=bi=ϵm∑i∂Cxi∂bi

We then pick another mini-batch from the remaining samples, repeat the above process and continue until the training set is exhausted. This is defined as one training *epoch*.

**Backpropegation**

To begin, we review and simplify the feed-forward component of DNN training to use matrix notation.

* wjkℓ, weight from neuron k in layer (ℓ−1) to neuron j in layer ℓ
* bjℓ, bias at neuron j in layer ℓ
* ajℓ, activation at neuron j in layer ℓ

∴ ajℓ=σ(∑kwjkℓak(ℓ−1)+bjℓ).

To convert to matrix format, define a weight matrix wℓ for layer ℓ where wℓ are weights connecting ***to*** layer ℓ's neurons.

wℓ=[wℓ[j,k]=wjkℓ]⏟k input weights}j neurons

Similarly for layer ℓ, bℓ is a bias vector with bjℓ the bias for neuron j in layer ℓ. aℓ is an activation vector with ajℓ the activation for neuron j in layer ℓ.

Finally, we use vectorization to apply the activation function to weights times previous layer activations plus biases.

σ([x1⋯xm])=[σ(x1)⋯σ(xm)]

Combining all of these, for layer ℓ

aℓ=σ(wℓa(ℓ−1)+bℓ)

The value zℓ=wℓa(ℓ−1)+bℓ is important, so we explicitly extract it and define it as the *weighted input* to neurons in layer ℓ. We can now write

aℓ=σ(zℓ)zjℓ=∑kwjkℓa(ℓ−1)+bjℓ

***Cost Function Assumptions***

The goal of backpropegation is to compute ∂c∂w and ∂C∂b for cost function C, w.r.t. weights w and biases b in the network. To do this, we make two assumptions about C. You can assume our C is MSE, C=12n∑x||y(x)=aL(x)||2, where

* n, total number of training examples;
* ∑x, sum over individual examples x;
* y=y(x), expected output;
* L, number of layers in network, and
* aL=aL(x), vector of activation outputs for input x.

1. Cost function C can be written as an average C=1n∑xCx over individual cost functions Cx for individual training examples x. This is required because backpropegation computes individual ∂Cx∂w and ∂Cx∂b, then recovers ∂C∂w and ∂C∂b by averaging the individual partial derivatives.
2. Cost function C can be written as a function of outputs from the neural network.

***Elementwise Product (Hadamard or Schur Product)***

For two vectors s and t, the elementwise product s⊙t is (s⊙t)=sjtj, *e.g.*,

[12]⊙[34]=[1⋅32⋅4]=[38]

This is know as the *Hadamard* or *Schur* product.

***Four Fundamental Backpropegation Equations***

At a basic level, backpropegation explains how changing w and b in a network changes C. Ultimately, this requires computing ∂C∂wjkℓ and ∂C∂bℓ. To do this, we introduce an intermediate quantity δjℓ, the error in neuron j in layer ℓ. Backpropegation computes δjℓ, then relates it to ∂C∂wjkℓ and ∂C∂bℓ.

Suppose, at some neuron, we modify zjℓ by adding a small change Δzjℓ. Now, the neuron outputs σ(zjℓ+Δzjℓ), propegating through follow-on layers in the network and changing the overall cost C by ∂C∂zjℓΔzjℓ. If ∂C∂zjℓ is large, we can improve C by choosing Δzjℓ with a sign opposite to ∂C∂zjℓ. If ∂C∂zjℓ is small, though, Δzjℓ will have little impact on C. Intuitively, ∂C∂zjℓ is a measure of the amount of error in a neuron.

Given this, we define error δjl of neuron j in layer ℓ as

δjℓ=∂C∂zjℓ

As before, δℓ is a vector of errors for neurons in layer ℓ. Backpropegation computes δℓ for every layer, then relates δℓ to the quantities of real interest, ∂C∂wjkℓ and ∂C∂bℓ

**Eq 1:** Computing δL error in the output layer.

Components of δL are

δjL=∂C∂ajLσ′(zjL)

∂C∂ajL measures how fast cost is changing as a function of output activation j. σ′(zyL) measures how fast activation function σ is changing at zyL. zyL is computed during feed-forward, so σ′(zyL) is easily obtained. ∂C∂ajL depends on C, but for our MSE cost function ∂C∂ajL= (ajL−yj).

To extend δjL to matrix-based form

δL=∇aC⊙σ′(zL)

where ∇aC is a vector whose components are ∂C∂ajL. ∇aC expresses the rate of change of C w.r.t. output activations. In our example ∇aC=(aL−y), so the full matrix form of Eq. 1 is

σL=(aL−y)⊙σ′(zL)

**Eq 2:** Computing δℓ from δℓ+1

Given δℓ+1

δℓ=((wℓ+1)Tδℓ+1)⊙σ′(zℓ)

where (wℓ+1)T is the transpose of the weight matrix wℓ+1 for layer ℓ+1.

Suppose we know the error δℓ+1 at layer ℓ+1. Applying (wℓ+1)T moves error backwards, giving us some measure of error in layer ℓ. Applying the Hadamard product ⊙σ′(zℓ) pushes error backwards through the activation function in layer ℓ, giving us backpropegated error δℓ in weighted input to layer ℓ.

We can use this to compute error δℓ for any layer ℓ by starting with δL (Eq. 1), using it to calculate δL−1, then δL−2 and so on until L−i=ℓ.

**Eq 3:** Rate of change of C w.r.t. bias

It turns out that ∂C∂bjℓ= δjℓ. That is, error δjℓ is exactly equal to the rate of change of bias ∂C∂bjℓ. Since we already know how to compute δjℓ, we can rewrite this as

∂C∂b=δ

where δ is being evaluated at the same neuron as bias b.

**Eq 4:** Rate of change of C w.r.t. weight

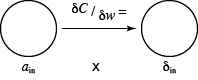
Here

∂C∂wjkℓ=akℓ−1δjℓ

so partial derivative ∂C∂wjkℓ depends on δℓ and aℓ−1, which we already know how to compute. Our equation can be rewritten as

∂C∂w=ainδout

where ain is activation of the neuron's input to its weight w and δout is the error of the neuron output for its weight w. Examining just weight w and the two neurons connected by that weight



***Backpropegation Algorithm***

Combining all four equations, we obtain

1. Input x, set a1 for input layer.
2. Feed-forward, for each ℓ=2,3,…,L compute zℓ=wℓaℓ−1+bℓ and a=σ(zℓ).
3. Output error δL, then compute vector δL=∇aC⊙σ′(zL).
4. Backpropegate error, for each ℓ=L−1,L−2,…,2 compute δℓ=((wℓ+1)Tδℓ+1)⊙σ′(zℓ).
5. Result  
   ∂C∂wjkℓ=aℓ−1δjℓ,∂C∂bjℓ=δjℓ

***Full Training Algorithm***

1. Input training examples.
2. For each training example x
   * set input activation ax,1,
   * feed-forward, for each ℓ=2,3,…,L compute zx,ℓ=wℓax,ℓ−1+bℓ and ax,l=σ(zx,l),
   * output error, compute δx,ℓ=∇aCx⊙σ′(zx,ℓ)
   * backpropegate error for each ℓ=L−1,L−2,…,2 by computing δx,ℓ=((wℓ+1)Tδx,ℓ+1)⊙σ′(zx,ℓ)
3. Gradient descent, for each ℓ=L,L−1,…,2 update weights and biases according to rules  
   wℓ→wℓ−ηm∑xδx,ℓ(ax,ℓ−1)Tbℓ→bℓ−ηm∑xδx,ℓ

**PyTorch**

PyTorch is, at its core, a tensor computing language with GPU acceleration support and a deep neural network library. PyTorch began as an internship program by Adam Paszke (now a Senior Research Scientist at Google) in October 2016. It was based on Torch, an open-source machine learning library and scientific computing framework written in the Lua programming language.

Three additional authors (Sam Gross, Soumith Chintala, and Gregory Chanan) formed the original author list. PyTorch was released as an open source machine learning language using the BSD open source license. Facebook currently operates both [PyTorch](https://ai.facebook.com/tools/pytorch/" \t "_blank) and the [Convolutional Architecture for Fast Feature Embedding](https://research.fb.com/downloads/caffe2/) (Caffe2). It is one of the standard libraries for deep neural network research and implementation.

***Tensors***

A basic data structure for holding data in PyTorch is a *tensor*. An *m × n* tensor is a multidimensional data structure with *n* rows and *m* columns, very similar to a Numpy ndarray. One critical difference between tensors and Numpy arrays is that tensors can be moved to the GPU for rapid processing. Below is a very simple example of creating a 2 × 2 PyTorch tensor.

> import torch > data = [ [1,2], [3,4] ] > t = torch.tensor( data ) > print( t ) tensor([[1, 2], [3, 4]])

If we wanted to move the tensor to the GPU, we would first check to ensure GPU processing is available, then use the to() method to transfer the tensor from CPU memory to GPU memory.

> if torch.cuda.is\_available(): > device = torch.device( 'cuda' ) > else: > device = torch.device( 'cpu' ) > t.to( device )

One important caveat is that you must do **all** processing on either the CPU or the GPU. You cannot split data structures and processing between the two processors. So, if you move your tensors to the GPU, you must also move your DNN models to the GPU and ensure all operations are performed on the GPU.

This is a [Jupyter notebook that implements a DNN](https://www.csc2.ncsu.edu/faculty/healey/msa/dnn/code/iris-dnn.ipynb) in the simplest possible manner. It is a good starting point for understanding what tensors are and how they can be created and manipulated. The DNN itself trains on six examples of three iris types (setosa, versicolor, virginica) using sepal and petal length and width.

**FCN Exercise 1: Wheat Seed Classification**

As an initial example, we will run a simple wheat seed classifier. The [dataset](https://www.csc2.ncsu.edu/faculty/healey/msa/dnn/code/wheat.csv) contains properties of three types of wheat seeds. The goal is to use these properties to predict the type of wheat the seed represents. we will first demonstrate a "from scratch" DNN that uses a simple single-layer neural network to train, then predict wheat seeds.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Province** | **Area** | **Perimeter** | **Compactness** | **Kernel Length** | **Kernel Width** | **Assymetry** | **Groove Length** | **Type** |
| Ontario | 15.26 | 14.84 | 0.871 | 5.763 | 3.312 | 2.221 | 5.22 | 1 |
| Manitoba | 14.88 | 14.57 | 0.8811 | 5.554 | 3.333 | 21.018/td> | 4.956 | 1 |
| Nova Scotia | 19.13 | 16.31 | 0.9035 | 6.183 | 3.902 | 2.109 | 5.924 | 2 |
| ⋯ | | | | | | | | |

The following Python-only DNN uses a simple single-layer 10-neuron DNN (ANN, actually) with a learning rate ϵ=0.1, a learning rate decay of 0.01 per epoch, 1000 training iterations per epoch, a single epoch, and five-fold cross validation during testing.

[Python Wheat DNN, Jupyter Notebook](https://www.csc2.ncsu.edu/faculty/healey/msa/dnn/notebook/simple-nn.ipynb)

Next, we'll show the processing the same dataset, but instead of doing it in raw Python, we'll use PyTorch, Facebook's Python-based DNN library. This will demonstrate how much easier it is to use PyTorch to build a significantly more complicated DNN to train, then predict wheat types based on wheat seed properties.

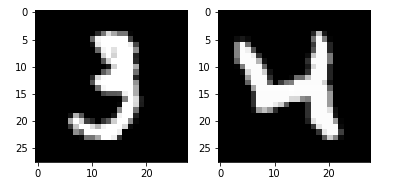
[Pytorch Wheat DNN](https://www.csc2.ncsu.edu/faculty/healey/msa/dnn/code/pytorch-nn.py)

The purpose of these examples is, first, to show you how to code your own DNN using basic Python, then how to use one of the most popular Python libraries (TensorFlow, programmed in C, is the other candidate) to perform the same computation in a simpler to program and more sophisticated, manner.

**FCN Exercise 2: Handwritten Number Recognition**

PyTorch's dataset repository includes the [MNIST database of handwritten images](https://en.wikipedia.org/wiki/MNIST_database), which includes [60,000 training examples](https://www.python-course.eu/data/mnist/mnist_train.csv) and [10,000 test examples](https://www.python-course.eu/data/mnist/mnist_test.csv). It is a very common dataset to use to test learning or pattern recognition algorithms on real-world data, without the need to hand-label a large training dataset.

Normally, images are processed with CNNs. However, we will use a simple FCN to recognize the handwritten images. This is done by converting each handwritten digit image into a set of pixel values, then converting that into a one-dimensional vector. This 1D vector acts as input to a single hidden layer, an ReLU (rectified linear unit) filter, and an output layer with ten possible classifications representing the ten digits 0–9. Here are two simple examples of the digits in the MNIST dataset.



Here is a Jupyter Notebook we will use to load the MNIST dataset, construct a simple FCN, then train and test it on the MNIST data.

[Digits FCN Jupyter Notebook](https://www.csc2.ncsu.edu/faculty/healey/msa/dnn/notebook/digit_ex.ipynb)

Even with a simple FCN with a single hidden layer and an ReLU filter, five training epochs (evaluating the training dataset five times) produces results of 97% or better on the test dataset.

**FCN Exercise 3: Image Processing**

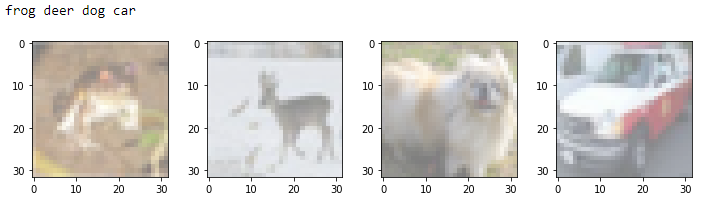
To see how well an FCN works on real two-dimensional images, we will work with the CIFAR10 dataset, also a part of PyTorch's dataset repository. CIFAR10 contains 50,000 training images and 10,000 test images of size 32×32×3: 32 pixels wide by 32 pixels high by 3 pixel components R (red), G (green), and B (blue).

train\_dataset = dsets.CIFAR10( root='./data', train=True, transform=xforms.ToTensor(), download=True ) test\_dataset = dsets.CIFAR10( root='./data', train=False, transform=xforms.ToTensor() ) classes = ( 'plane','car','bird','cat','deer','dog','frog','horse','ship','truck' )

Note the classes variable. This is used to convert the label value for an image into a semantic text description of the class it belongs to. You will probably want to define this and index into it to better understand which classes the images belong to.

If you want to examine some of the images in the CIFAR10 dataset, you can modify the code in our original FCN example. However, a simpler way to do this and show more images would be as follows.

# Let's look at four of the images in the training dataset, and the # corresponding label (which defines the object the image represents) # Grab the first four images and corresponding objects they represent # from the training set img = [] val = [] for i in range( 4 ): im, v = train\_dataset[ 20 \* i ] img.append( im ) val.append( v ) img[ i ] = img[ i ] / 2 + 0.5 img[ i ] = img[ i ].numpy() img[ i ] = img[ i ].transpose( 1, 2, 0 ) # Print out what objects the four images are meant to represent print( ' '.join( classes[ val[ i ] ] for i in range( 4 ) ) ) # Create a single row of four images fig, axes = plt.subplots( 1, 4, figsize=( 12, 2.5 ) ) for i in range( 4 ): axes[ i ].imshow( img[ i ] )



Apart from changing any occurrences 28 \* 28 to 32 \* 32 \* 3 (to account for the different input size), the remainder of the code can be identical to the MNIST FCN. You're certainly encouraged to also vary things like criteria, optimizer, hidden\_size, the number of hidden layers, and other properties of the FCN to try to improve performance. Remember that, for ten classes, just like the MNIST dataset, chance is 10%. You're unlikely to obtain accuracies anywhere near the 97% we produce for the MNIST images, but you should be able to do much better than 10%, even for this most simple FCN.

Here is a simple Jupyter Notebook that trains on the CIFAR10 dataset using our simple FCN.

[CIFAR FCN Jupyter Notebook](https://www.csc2.ncsu.edu/faculty/healey/msa/dnn/notebook/CIFAR_FCN_ex.ipynb)

**CNN Exercise 4: Image Processing**

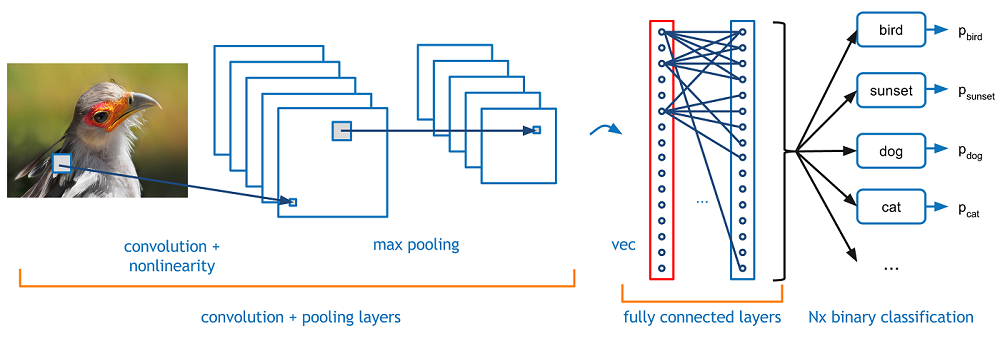
The wheat dataset is fairly easy to process, but it is probably more representative of the type of task you need to solve: classification from one or more input properties. DNNs themselves have been applied most successfully to image data using convolutional neural networks (CNNs). A CNN converts an n×m image to an nm vector, then uses that vector as input into a convolution stage. Here, a collection of k *kernels* are convolved against the pixels and their immediate neighbours to produce scalar values. For each kernel, a column of nm convolved values are created, one for each pixel in the image. In the simplest CNN, the column values are evaluated to produce a single, representative value. For example, *max* scans a column and extracts the largest value. These values form a k-length *input vector* to a follow-on fully connected network. This network processes output from the CNN to generate a final classification prediction.

**Convolutional Neural Networks**

Convolutional neural networks (CNNs) extend FCNs by preceding the fully connected layers with a set of convolutional layers. CNNs are commonly used to analyze images, although [recent research](https://arxiv.org/pdf/1803.01271v2.pdf) has shown that they can handle other data modalities like text with excellent performance.

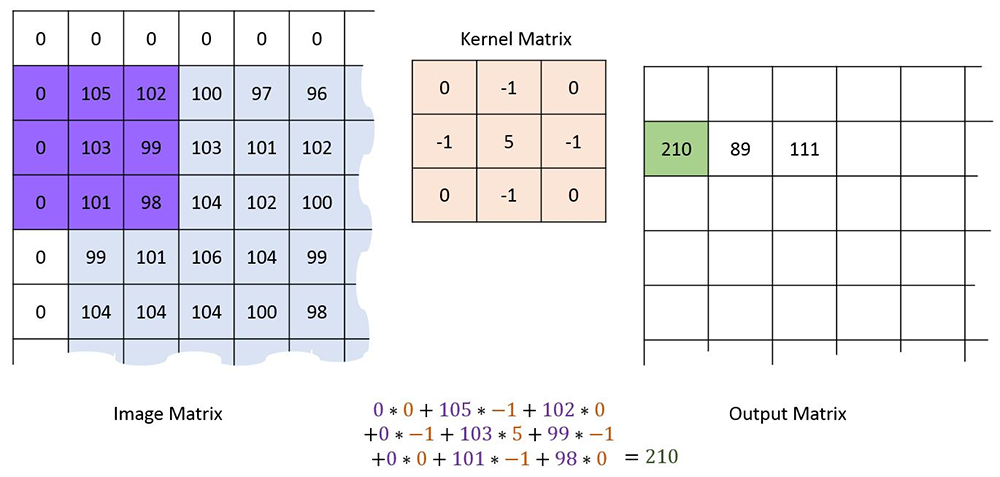
CNNs are made up of a number of standard operations to produce new, hidden layers. These include

* **Convolution (CONV):** Application of a *convolve* operation across a layer, producing a new layer with the convolved results
* **Filter (RELU):** Application of a filter like ReLU to values in a layer, produce a new layer with filtered results
* **Pooling (POOL):** Application of a pooling (aggregation) operator to values in a layer, produce a new, smaller layer with pooled results



In practice, it is common to apply a series of CONV-RELU layers, follow them with a POOL layer, and repeat this pattern until an image has been processed to a small size. At this point, results are formed into a 1D vector which acts as input to an FCN. The FCN produces a set of probabilities for each possible classification (softmax).

***Convolution***



Convolution combines a pixel in an image and its neighbours by placing a *kernel* of a given size centered over the pixel, then multiplying the corresponding pixel and kernel values, and summing them together to produce a final filtered result.

The example above using a 3×3 kernel with values -1, 0, and 5 at various positions within its nine cells. When we center it over the pixel at the center of the purple box, multiple, and sum, we obtain a final filtered value of 210 for that position in the image. Kernels are designed to identify specific properties of an image, producing large values when those properties are located, and small values when they are not. For example, the Kirsch filter is designed to identify edges in an image. Convolution of a simple animation with a Kirsch filter produces the result shown below.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | |  |  |  | | --- | --- | --- | | 5 | 5 | 5 | | -3 | 0 | -3 | | -3 | -3 | -3 | | Kirsch kernel | | | |
|  |

Normally, we use numerous kernels to identify different properties or *features* in an image. Each kernel produces a *feature map* from the image. These feature maps are normally stacked one on top of another, producing a result with width and height one less than the original image size, and depth equal to the number of kernels applied to the image. How does the CNN decide on the kernel values for each kernel? This occurs during training. Kernel values are initially random, and slowly converge along with edge weights during backpropagation to identify image properties salient to classifying the images.

***Filtering***

Filtering adjusts values in a feature map, for example, by normalizing them, or by removing negative values. The common ReLU filter, for example, removes negative values and retains positive values.

|  |  |
| --- | --- |
| Original | Filtered |
| Filtered w/ReLU |

***Pooling***

Pooling takes a feature map, and reduces its size by aggregating block of values. Aggregation can use any common mathematical operator like average, median, maximum or minimum. Pooling uses a *window size* which defines its width and height, and a *stride* which defines the step size it uses as it slides over the feature map values it pools.

|  |  |  |
| --- | --- | --- |
|  | max pooling →   2×2 window   → stride 2 |  |

**Images**

Recall in the previous discussion of FCNs, we used the MNIST handwriting image dataset. We are now switching to the CIFAR10 dataset of photographic images. In your exercise, you were asked to re-purpose the FCN to handle images from CIFAR10. Here is a simple Jupyter Notebook that does that.

[CIFAR FCN Jupyter Notebook](https://www.csc2.ncsu.edu/faculty/healey/msa/dnn/notebook/CIFAR_FCN_ex.ipynb)

Although this produces results better than chance, we can further improve performance by using a simple CNN.

[CIFAR CNN Jupyter Notebook](https://www.csc2.ncsu.edu/faculty/healey/msa/dnn/notebook/CIFAR_CNN_ex.ipynb)

Results for the CNN are indeed better than the FCN, but only by a few percentage points. This suggests the CNN could be improved possibly significantly, by expanding the CONV/RELU/POOL part of the image to better capture the features needed to differentiate different image classes from one another. One clue is in the individual class accuracies, which show that natural images like cats and deer are being labeled much less accurately than man-made images like cars and trucks.

**Recurrent Neural Networks**

Recurrent neural networks (RNNs) are normally used to handle sequence-based data I={i1,…,in}. In simple terms, a basic DNN is designed. The first sample in the input sequence i1 is fed into the DNN, producing both an output o1 and a hidden output h1, o1=h1. The output can be fed into a standard FCN is the user wants to use it for classification. The hidden output h1 is combined with the next sample in the input sequence i2, and this pair (h1,i2) is used as input into the DNN. This produces another output o2 and hidden output h2, o2=h2. The process continues until all samples in the input sequence are processed, producing a final output on and hidden output hn. In other words, a single DNN processes input samples and hidden output from the previous step *recursively* or *recurrently*, generating a result that represent both the output from the current RNN step and a hidden output for the next RNN step.

Note that one detail was left unspecified: if the DNN expects a sample input and previous hidden output pair as input, what is the hidden output for the first step in the recursion? Normally, a random hidden output h0 is generated and used for the first processing step in an RNN.

Visually, these images from Michael Phi's excellent [page on LSTMs and GRUs](https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21) shows clearly how input samples ij from the input sequence I are fed into the RNN's DNN structure one-by-one.

|  |
| --- |
|  |
| Processing input samples one-by-one from input sequence I (attribution: [Michael Phi)](https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21) |

This close-up image shows how the hidden output hj−1 and the current input ij are combined and passed through a tanh function to produce a continuous output oj and hidden output value hj.

|  |
| --- |
|  |
| A recurrent step combines previous hidden input hj−1 and current input ij, then passes the combination through a tanh function to produce a continuous output oj and hidden output hj from the DNN (attribution: [Michael Phi)](https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21) |

One way of understanding RNNs is that they use the concept of *sequence memory*. Over time, they have the ability to "learn" sequential patterns. In theory, this is something an FCN or CNN would have more difficulty to do, since an RNN uses previous information (hidden output) where FCNs and CNNs do not. Programatically, this can be expressed in the following way.

rnn = RNN() fcn = FeedForwardDNN() hidden = random() for i in input: output,hidden = rnn( i, hidden ) prediction = fcn( output )

***Vanishing Gradients***

Although basic RNNs are designed to "remember" previous results, they do have an important problem. As an RNN processes inputs, it begins to forget what it has seen in previous steps. Intuitively, you could say that a basic RNN has a short-term memory, but not a long-term memory. This happens because of the way backpropegation occurs. To understand this, think of how any DNN works. There are three major steps: (1) feedforward; (2) error based on predicted class; (3) backpropegation of error based on gradient descent to adjust edge weights. At any given layer in the DNN, its gradient values depend on the successive layer's gradient values. So, if the following layer has small gradient values, the current layer's gradients will be even smaller. This is the vanishing gradient effect. Layers near the beginning of the DNN are learning little or nothing.

Consider an RNN, where each step in the RNN is analogous to a layer in a DNN. A method similar to backpropegation (backpropegation over time) is used to improve the network, so the vanishing gradient effect occurs over steps in the RNN, as opposed to layers in a DNN. This explains intuitively why an RNN can remember recent timesteps, but not ones further back in time.

What is the practical effect of the lack of long-term memory? Consider an RNN processing a sentence term by term.

My dog Goro and I went for a walk, but a squirrel ran in front of us and he started chasing it.

It's easy for us to recognize that "he started chasing it" refers to Goro. But an RNN may no longer remember that "he" refers to a dog named Goro. This is a serious disadvantage for a type of DNN specifically designed to process sequence data.

**Long Short-Term Memory RNNs**

Long short-term memory (LSTM) DNNs are a type of RNN designed to address the vanishing gradient problem. Initially introduced by Hochreiter & Schmidhuber in 1977, LSTMs use a set of three "gates" to maintain both a long-term and a short-term memory. Below is an image of the internals of an LSTM.

|  |
| --- |
|  |
| The gates, activation functions, and data flow through the DNNs used in LSTMs (attribution: [Michael Phi)](https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21) |

To understand what happens internally within the LSTM, we can look at the various data paths. An LSTM is made up of three pieces of information: the current input value from the sequence ij, the previous hidden state hj−1, and the previous *cell state* cj−1. Each iteration of an LSTM RNN produces a new hidden state hj, and new cell state cj, and an output oj, with hj=oj as before. Intuitively, we think of the hidden state as our short-term memory, and the cell state as our long-term memory.

The path along the top manages the cell state cj−1. As noted above, the cell state is the long-term "memory" of the network over the sequence processed to date. We must decide what to remove or "forget" from the cell state. Not surprisingly, this is called the *forget gate layer*. The previous hidden state hj−1 and current input ij are used to control this decision. Mathematically, the output combines the hidden and input values, then passes them through a sigmoid operator.

f=σ(Wf⋅[hj−1,ij]+bf)

The previous hidden state and current input value [hj−1,ij] are multiplied by a weight in Wf and combined with a bias in bf, then passed through the σ function to transform the result to the continuous range [0…1]. A value close to 0 means completely forget, and a value close to 1 means fully retain. Remember that we are (normally) working with vectors hj−1, ij, Wf, and bf. The resulting vector f is pointwise-multiplied to the previous cell state cj−1 using the elementwise product operator ⊙ to alter its values. At this point it should be clear why values close to 0 or close to 1 remove or retain information in cj−1.

Next comes the *input gate*, which decides what new information will be stored in long-term memory cj. As with the forget gate, it combines the previous hidden state hj−1 and the current input value ij to make this decision. This is made up of two parts i1 and i2.i1=σ(Wi1⋅[hj−1,ij]+bi1)i2=tanh(Wi2⋅[hj−1,ij]+bi2)

Intuitively, i1 dictates what information "passes through" to the cell state, again with 0 indicating no pass-thru, and 1 indicating complete pass-thru. i2 is more difficult to explain intuitively, but its purpose is to regulate the network. i1 and i2 are multiplied, and the result is pointwise-added to the cell state after the forget gate is applied. Again, at this point it should be clear how values on the range [0…1] are affecting what new information is being added into long-term memory.

The final stage is the *output gate*, which combines all three pieces of information: the previous hidden state hj−1 (short-term memory), the current input value ij (new input), and the new cell state after the forget and input gates are applied, cj (long-term memory) using two parts o1 representing a combination of the previous hidden state and the current input value and o2 representing long-term memory.

o1=σ(Wo1⋅[hj−1,ij]+bo1)o2=tanh(Wo2⋅cj+bo2)hj,oj=o1⊙o2

The new short-term memory o1 and long-term memory o2 are pointwise-multiplied to produce the new hidden state and output hj and oj for the current sequence entry.

As with all DNNs, the key on how the network functions lies in its weights and biases Wf, bf, Wi1, bi1, Wi2, bi2, Wo1, bo1, Wo2, and bo2. And as with all DNNs, these values are updated during each step of DNN training by using gradient descent and optimization to adjust the weights and biases in directions that produce results with a lower loss or error.

One final note about LSTMs is that they generally come in two forms: *many-to-many* or *many-to-one*. The difference is in whether we care about intermediate output values, or only the final output value produced after the last input in in the sequence is processed. For example, if we are performing text generation, we would use a many-to-many approach. The output from each step would be fed through an FCN to produce a generated term, appended to the terms to date as we build up a sentence related to the input being processed. If we were performing sentiment analysis, we would most likely use a many-to-one approach, where we ignored intermediate outputs and passed only the final output through an FCN to determine a sentiment for the input sequence as a whole.

***Airline Passenger Example***

As a "simple" example of an LSTM, we'll use a built-in dataset from [seaborn](https://seaborn.pydata.org/), a Python statistical graphics and visualization library. seaborn includes a airline flight dataset with 144 months of data that includes, among other things, the number of passengers that flew each month. We will use an LSTM RNN to train on twelve months (one year) of passenger data, then use the resulting model to predict the number of passengers flying the month immediately following our twelve month training period.

* [Flight LSTM Jupyter Notebook](https://www.csc2.ncsu.edu/faculty/healey/msa/dnn/notebook/flight.ipynb)
* [PyTorch Flight LSTM](https://www.csc2.ncsu.edu/faculty/healey/msa/dnn/code/flight.py)

import seaborn as sns flight\_data = sns.load\_dataset( 'flights' ) print( flight\_data.head() ) print( len( flight\_data ) ) year month passengers 0 1949 January 112 1 1949 February 118 2 1949 March 132 3 1949 April 129 4 1949 May 121 144

To start, we load the libraries we will use in our program.

import matplotlib.pyplot as plt import numpy as np import pandas as pd import seaborn as sns import torch import torch.nn as nn from sklearn.preprocessing import MinMaxScaler

Next, we will create our LSTM neural network. Since this is a simple example, the LSTM will be made of of a recurrent node as described above, with output from the node fed into a fully-connected network with a single hidden layer.

class LSTM( nn.Module ): def \_\_init\_\_( self, input\_size=1, hidden\_n=100, output\_size=1 ): super( LSTM, self ).\_\_init\_\_() self.hidden\_layer\_size = hidden\_n self.lstm = nn.LSTM( input\_size, hidden\_n ) self.fcn = nn.Linear( hidden\_n, output\_size ) # Hidden cell contains previous hidden state, previous cell state, # randomized to start self.hidden\_cell =\ ( torch.zeros( 1, 1, self.hidden\_layer\_size ), torch.zeros( 1, 1, self.hidden\_layer\_size ) ) # End method \_\_init\_\_ def forward( self, input\_seq ): # input\_seq is a 12-value tensor, the 12 months of passengers to # train on, normalized on the range -1..1, as a single row # Input to an LSTM is of shape (seq\_len, batch, input\_size) # so view below creates a column of 12 "samples", each sample is # a single value, a batch size of 1, and a length of 1 (one 12-value # sample sequence) input\_seq = input\_seq.view( len( input\_seq ), 1, 1 ) # Ask the LSTM for the output and the (hidden,cell) state based # on the 12-value input and the current (hidden,cell) state, this # will recurse the LSTM 12 times lstm\_out,self.hidden\_cell = self.lstm( input\_seq, self.hidden\_cell ) lstm\_out = lstm\_out.view( len( input\_seq ), -1 ) # Run final output through the LSTM's FCN to get class # probabilities predictions = self.fcn( lstm\_out ) # Highest probability is the class we estimate return predictions[ -1 ] # End method forward # End class LSTM

We will also create a helper function that takes a full sequence of monthly passenger data, and breaks it into tuples of training and label data. The training data is a list of twelve months of passenger counts ti={pj,…,pj+11}. The label is the passenger count for the month immediately following the twelve-month training period li=pj+12. Each of these (ti,li) tuples are stores in a list of training samples used to train our LSTM T={t0,t1,…,t119}.

def create\_IO\_seq( input, tw ): # Create a set of time series to process during training, input is the # entire data stream to divide, tw is time window size in samples # # IO\_seq is a (train\_seq,label) tuple list, train\_seq is a 12-month # set of passengers, label is a single passenger count following the # 12-month sequence IO\_seq = [ ] n = len( input ) for i in range( 0, n - tw ): # Grab tw elements as training sequence, next element that follows # is the label (i.e., the number of passengers following the given # 12-month period) train\_seq = input[ i: i + tw ] # To be pedantic, pull single tensor value as float, then make a # single-element float list and convert it back to a tensor; can be # done in a single step as: # # train\_label = input[ i + tw: i + tw + 1 ] # # but I find that harder to understand # # val is a single float, the number of passengers following the # 12-month training sequence we just extracted # # train\_label is [ val ] (a single-element float list) as a tensor val = input[ i + tw ].item() train\_label = torch.FloatTensor( [ val ] ) IO\_seq.append( (train\_seq,train\_label) ) return IO\_seq # End function create\_IO\_seq

Next, we will load our dataset, transform the values from raw passenger counts into "normalized" values on the range [0…1], then use our helper function create\_IO\_seq to create our training sequence.

# Load data, divide into train and test flight\_data = sns.load\_dataset( 'flights' ) # First 132 months for train, last 12 months for test, split into # 12-month training sequences and 1-value labels: # # 1 2 3 4 5 6 7 8 9 10 11 12 13 # - 12 months of training - next value is label # so train on 12-month sequence..then see how many passengers next month all\_data = flight\_data[ 'passengers' ].values.astype( float ) test\_data\_size = 12 train\_data = all\_data[ :-test\_data\_size ] test\_data = all\_data[ -test\_data\_size: ] # Transform/normalize data to range -1..1 scaler = MinMaxScaler( feature\_range=(-1,1) ) # First reshape data into a single column then transform to range -1..1 train\_data\_norm =\ scaler.fit\_transform( train\_data.reshape( len( train\_data ), 1 ) ) # Convert back to PyTorch tensor that's a single row train\_data\_norm =\ torch.FloatTensor( train\_data\_norm ).view( len( train\_data\_norm ) ) # Create 12-value training sequences and correspond next value label train\_window = 12 train\_IO\_seq = create\_IO\_seq( train\_data\_norm, train\_window )

Finally, we can create our LSTM RNN, choose our loss and optimizer functions (mean squared error and Adam, respectively), and use our training sequence to teach the model how to predict future passenger counts based on the previous year's monthly passenger count sequences. We will process the entire training sequence for 150 epochs.

# Train LSTM model.train() epochs = 150 for i in range( 0, epochs ): for seq,label in train\_IO\_seq: optimizer.zero\_grad() # Re-initialize hidden and cell state to random prior to # walking over the samples in the training sequence model.hidden\_cell =\ ( torch.zeros( 1, 1, model.hidden\_layer\_size ), torch.zeros( 1, 1, model.hidden\_layer\_size ) ) y\_pred = model( seq ) single\_loss = loss\_function( y\_pred, label ) single\_loss.backward() optimizer.step() if i % 25 == 1: print( f'epoch {i:3}; loss: {single\_loss.item():10.8f}' ) print( f'epoch {i:3}; loss: {single\_loss.item():10.10f}' )

Notice an important subtlety here. The order of processing for our LSTM is:

1. Zero any previous gradient information.
2. Randomize both our hidden and cell states.
3. Run the LSTM over **the entire** 12-month sequence, storing the final output value as y\_pred.
4. Compute the loss between our prediction y\_pred and the known passenger count for the 13th month.
5. **Now** use gradient descent to backpropegate the loss through the LSTM.
6. Use the Adam optimizer to update the LSTM's weights, hopefully to better predict future passenger counts.

Critically, gradient descent, backpropegation, and weight optimization happen after each 12-month sequence is processed, and **not** after each month in the 12-month sequence is processed. This detail is important to understand.

Once the network is trained, we can predict passenger counts for the final twelve months, and compare them to the known counts. Notice we have two options for testing. We can take the *known* 12-month sequences, or we can take a combination of the *known* and *predicted* values to form a 12-month sequence.

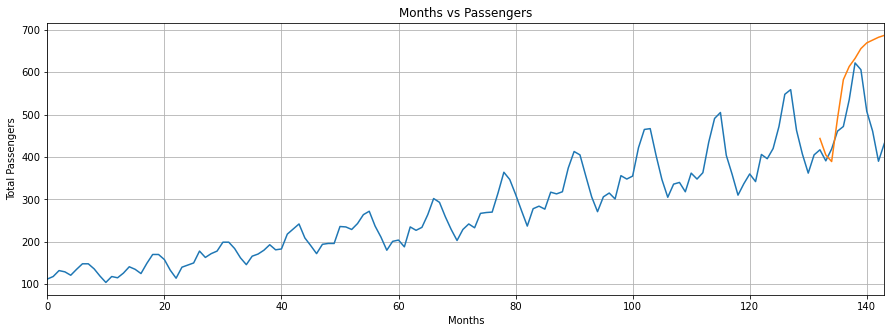
t1={p121,p122,…,p132}→l1t2={p122,p123,…,p133}→l2⋯t12={p132,p133,…,p143}→l12versust={p121,p122,…,p132}→l1t={p122,p123,…,l1}→l2⋯t={p132,l1,…,l11}→l12

We choose to implement the second approach. This approach would be required if you are using all of your known data to predict a target variable different from your predictors. For example, suppose we had 144 months of temperature data, and 156 months of precipitation data. If we use the full 144 months of temperature data to build our model, then the final 12 months we predict for testing have no corresponding temperature data. In this case, we have no choice but to use our predictions to "fill in" the unavailable temperature data as we predict the 1st, 2nd, …, and 12th precipitation values. The tradeoff is more data during training versus estimated data that likely contains errors during testing for accuracy.

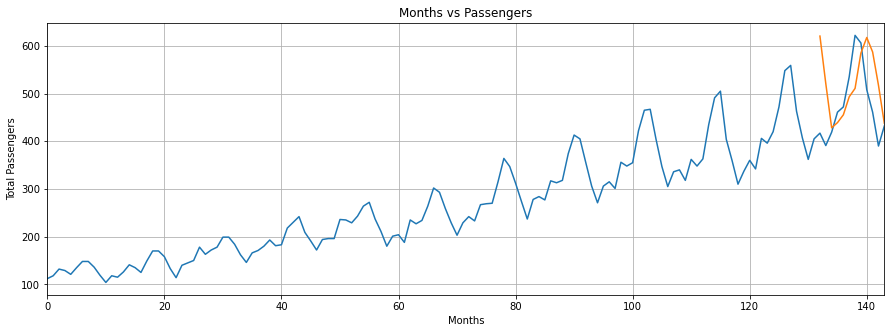
# Predict final twelve months model.eval() fut\_pred = 12 # Grab last twelve months of data, this will be the sequence used to # predict the first test value (remember, 132 training and 12 test # values were split at the beginning of the program) test\_outputs = [ ] test\_inputs = train\_data\_norm[ -train\_window: ].tolist() # Run through all 12 test values for i in range( 0, fut\_pred ): # Convert list of last 12 (normalized) passengers to a tensor seq = torch.FloatTensor( test\_inputs[ -train\_window: ] ) # with torch.no\_grad() runs LSTM without calculating gradients, we # can only do this b/c we know we don't need gradients, backwards() # is not called at the end of this training run, b/c we are passing # one single 12-value sequence and only care about the final output with torch.no\_grad(): model.hidden =\ ( torch.zeros( 1, 1, model.hidden\_layer\_size ), torch.zeros( 1, 1, model.hidden\_layer\_size ) ) # Append output of LSTM to test\_inputs, so when we loop again and # grab the last 12 values for input, it includes the output(s) # the LSTM is generating. Also, make sure to save the outputs for # later accuracy calculations test\_inputs.append( model( seq ).item() ) test\_outputs.append( test\_inputs[ -1 ] )

Finally, to visually inspect our predictions, we plot the full 144-month sequence of known passenger counts in blue, then show the last twelve months of predicted passenger counts in orange.

# The LSTM output is normalized on range -1..1, so we need to invert this # to get actual passenger numbers, need these to do a proper comparison to # known passenger numbers for accuracy calculations actual\_pred =\ scaler.inverse\_transform( np.array( test\_outputs ).reshape( -1, 1 ) ) # Plot known values in blue fig\_size = plt.rcParams[ 'figure.figsize' ] fig\_size[ 0 ] = 15 fig\_size[ 1 ] = 5 plt.rcParams[ 'figure.figsize' ] = fig\_size plt.title( 'Months vs Passengers' ) plt.ylabel( 'Total Passengers' ) plt.xlabel( 'Months' ) plt.grid( True ) plt.autoscale( axis='x', tight=True ) plt.plot( flight\_data[ 'passengers' ] ) # Add in the predicted values in orange x = np.arange( 132, 144, 1 ) plt.plot( x, actual\_pred ) plt.show()



The results are acceptable but not outstanding, although the LSTM was able to catch the up–down seasonal variation in the data, something a basic RNN would probably miss. You can compare this to an approach where we use only known values during testing. Not surprisingly, this produces slightly better results.



**DNN Practice Exercises**

Below is an example of the MNIST problem solved using a single-layer FCN ([Jupyter Notebook](https://www.csc2.ncsu.edu/faculty/healey/msa/dnn/notebook/MNIST-FCN.ipynb), [Python file](https://www.csc2.ncsu.edu/faculty/healey/msa/dnn/code/MNIST-FCN.py)). Extend this example to use two hidden layers instead of one. The second hidden layer should take input from the first, contain 64 nodes, and use ReLU to transform its output to be continuous. **Note:** to achieve this, you should **only** need to make minor changes in the FCN class's init() and forward() methods.

# Perform all required imports import numpy as np import torch import torch.nn as nn import torch.optim as optim import torchvision.datasets as datasets import torchvision.transforms as transforms class FCN( nn.Module ): # FCN model to convert image to digit def \_\_init\_\_( self ): super( FCN, self ).\_\_init\_\_() self.hidden0 = nn.Linear( 784, 128 ) self.output = nn.Linear( 128, 10 ) self.sigmoid = nn.Sigmoid() self.softmax = nn.LogSoftmax( dim=1 ) # End function \_\_init\_\_ def forward( self, input ): hidden = self.hidden0( input ) hidden = self.sigmoid( hidden ) output = self.output( hidden ) output = self.softmax( output ) return output # End function forward # End class FCN def train( model, epoch, trainloader ): # Training function, use stochastic gradient descent to optimize criterion = nn.NLLLoss() optimizer = optim.SGD( model.parameters(), lr=0.003, momentum=0.9 ) for e in range( 0, epoch ): running\_loss = 0 for images,labels in trainloader: # Flatten images into a 784-long vector images = images.view( images.shape[ 0 ], -1 ) optimizer.zero\_grad() output = model( images ) loss = criterion( output, labels ) loss.backward() optimizer.step() running\_loss += loss.item() print( 'Epoch {}, training loss: {}'.\ format( e, running\_loss / len( trainloader ) ) ) # End function train # Mainline # Transform incoming greyscale data to a tensor, the mean and stdev of # the MNIST images is 0.1307 and 0.3081, so use Normalize to convert # this to a range of -1..1 (NB: mean and stdev can take up to 3 arguments, # to support different mean/stdev for R,G,B image channels, but a greyscale # image only has one channel) transform = transforms.Compose( [ transforms.ToTensor(), transforms.Normalize( (0.5,), (0.5,) ) ] ) # Download datasets, if not already downloaded trainset =\ datasets.MNIST( './', download=True, train=True, transform=transform ) testset =\ datasets.MNIST( './', download=True, train=False, transform=transform ) # Process data in batches of 64 items, both for training and testing trainloader =\ torch.utils.data.DataLoader( trainset, batch\_size=64, shuffle=True ) testloader =\ torch.utils.data.DataLoader( testset, batch\_size=64, shuffle=True ) model = FCN() # Put the model in training mode, train for 15 epochs model.train() train( model, 15, trainloader ) # Evaluate the model on the 10000 test images n = 0 correct = 0 model.eval() for images,labels in testloader: for i in range( 0, len( labels ) ): img = images[ i ].view( 1, 784 ) prob = model( img ) # Invert probabilities since they are natural log'd # (LogSoftmax), then from a tensor to a list prob = torch.exp( prob ) prob = prob[ 0 ].tolist() pred\_label = prob.index( max( prob ) ) true\_label = labels[ i ].item() if true\_label == pred\_label: correct += 1 n += 1 print( f'Images tested: {n}' ) print( f'Accuracy: {correct / n \* 100.0:.2f}%' )

**MNIST FCN Solution**

Normally, image data is processed with a CNN rather than an FCN. This worked well here because taking the 2×2 image and concatenating rows to create a single vector of greyscale values formed patterns that distinguished different digits well. Could we do better with a CNN? Would you like to find out? If so, try implementing one.

This is more complicated than the previous example, because you will need to completely replace the FCN class with a CNN class. The rest of the code can remain unchanged, except for: (1) using a different criterion better suited for images, (2) no need to flatten images because a CNN can handle 2D images directly, unlike an FCN which expects its input to be single 1D vector, and (3) the evaluation section of the mainline, which will need to account for the format of the output from the CNN model.

Here is the CNN model structure you should use.

1. A 2D convolution that takes the MNIST images as input and produces six filters using a 4×4 kernel with a stride of 1.
2. An ReLU activation on the results to scale them to the range [0…1].
3. A 2D maxpooling of size 2×2 to downsample the image from size 28×28. **Be careful here!** Think hard about what size of image results you'll get from your 2D covolution step (hint: it **IS NOT** 28×28, consider what happens when a 4×4 filter reaches either the right edge or bottom edge of a 28×28 image). You will need to add padding to the MaxPool2D operation to account for this.
4. Another 2D convolution producing 16 filters using a 2×2 kernel with a stride of 1.
5. An ReLU activation on the results.
6. A 2D maxpooling of size 2×2.
7. Flatten the results into a single 1D vector as input for the follow-on FCN.
8. A linear layer with 120 nodes taking the results of the convolution stage as input.
9. An ReLU activation on the results.
10. A linear layer with 10 nodes taking output from the previous hidden layer and producing probabilities for the ten possible handwritten letter types.
11. Finally, a LogSoftmax operation on the probabilities to produce normalized logarithmic results.
12. Update your criterion to use nn.CrossEntropyLoss() rather than nn.NLLLoss() in your train() function.
13. In your train() function, you **DO NOT** need to flatten the images with view(), since a CNN takes the 2D 28×28 greyscale images directly.

**MNIST CNN Solution**

In my testing, results from the CNN were 98% or higher, slightly better than the FCN example.

Updated 02-Apr-21