Deep Learning for Human Beings

Understand How Deep Neural Networks Work

By Mohit Deshpande

Table Of Contents

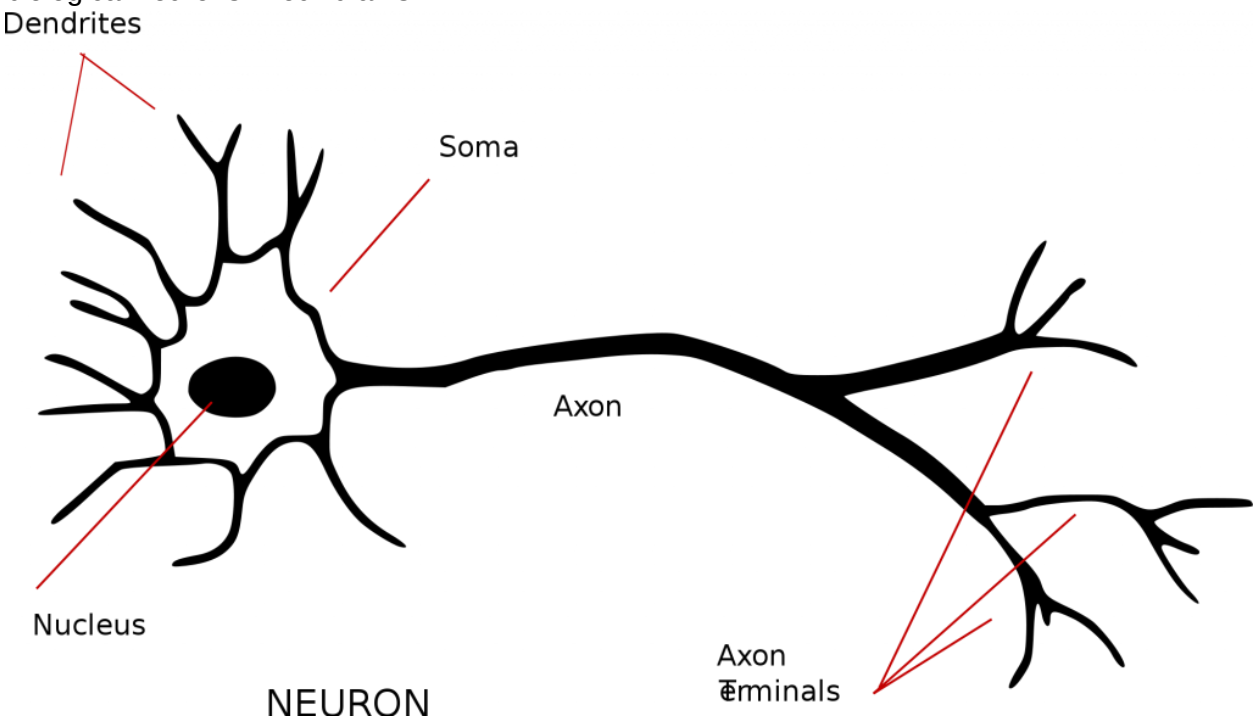
Perceptrons: The First Neural Networks

Neural Networks have become *incredibly* popular over the past few years, and new architectures, neuron types, activation functions, and training techniques pop up all the time in research. But without a fundamental understanding of neural networks, it can be quite difficult to keep up with the flurry of new work in this area.

To understand the modern approaches, we have to understand the tiniest, most fundamental building block of these so-called deep neural networks: the **neuron**. In particular, we'll see how to combine several of them into a **layer** and create a neural network called the **perceptron**. We'll write Python code (using numpy) to build a perceptron network from scratch and implement the learning algorithm. For the completed code, download the ZIP file here.

Biological Neurons

Perceptrons and artificial neurons actually date back to 1958. Frank Rosenblatt was a psychologist trying to solidify a mathematical model for biological neurons. To better understand the motivation behind the perceptron, we need a superficial understanding of the structure of biological neurons in our brains.



(Credit: https://commons.wikimedia.org/wiki/File:Neuron_-_annotated.svg)

Let's consider a biological neuron. The point of this cell is to take in some input (in the form of electrical signals in our brains), do some processing, and produce some output (also an electrical signal). One very important thing to note is that the inputs and outputs are *binary* (0 or 1)! An individual neuron accepts inputs, usually from other neurons, through its **dendrites**.

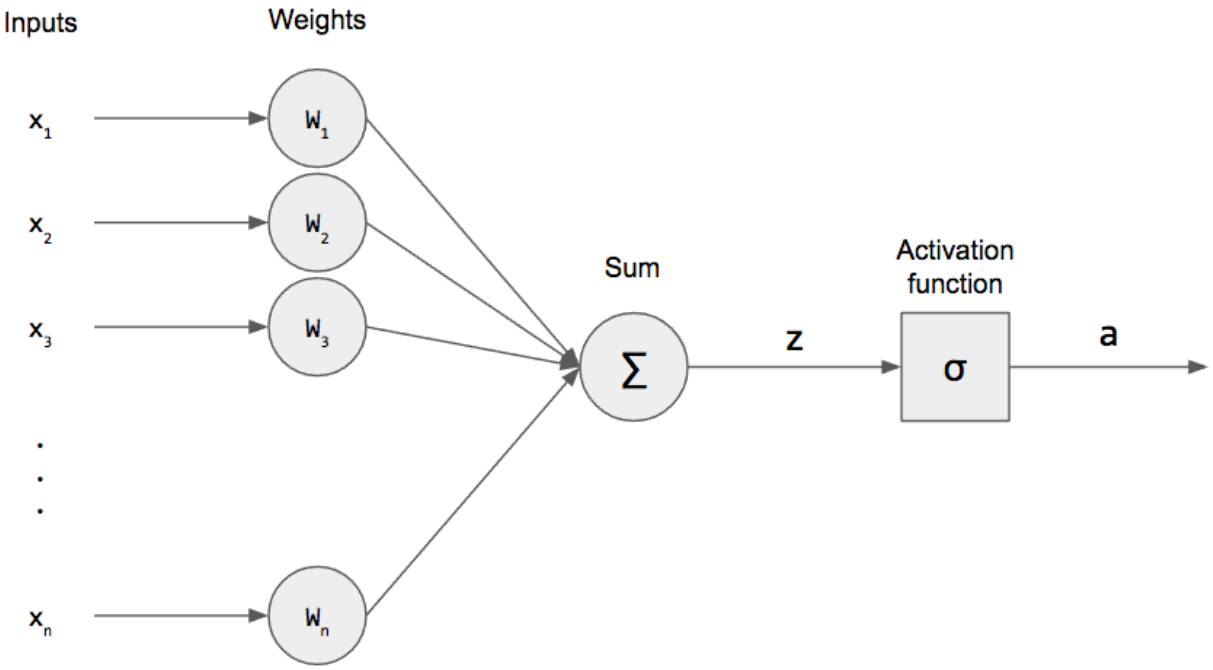
Although the image above doesn't depict it, the dendrites connect with other neurons through a gap called the **synapse** that assigns a weight to a particular input. Then, all of these inputs are considered together when they are processed in the cell body, or **soma**.

Neurons exhibit an all-or-nothing behavior. In other words, if the combination of inputs exceeds a certain threshold, then an output signal is produced, i.e., the neuron "fires." If the combination falls short of the threshold, then the neuron doesn't produce any output, i.e., the neuron "doesn't fire." In the case where the neuron *does* fire, the output travels along the **axon** to the **axon terminals**. These axon terminals are connected to the dendrites of other neurons through the synapse.

Let's take a moment to recap biological neurons. They take some binary inputs through the dendrites, but not all inputs are treated the same since they are weighted. We combine these weighted signals and, if they surpass a threshold, the neuron fires. This single output travels along the axon to other neurons. Now that we have this summary in mind, we can develop mathematical equations to roughly represent a biological neuron.

Artificial Neurons

Now that we have some understanding of biological neurons, the mathematical model should follow from the operations of a neuron.



In this model, we have n binary **inputs** (usually given as a vector) and exactly the same number of **weights** $W_1, ..., W_n$. We multiply these together and sum them up. We denote this as z and call it the **pre-activation**.

$$z = \sum_{i=1}^{n} W_i x_i = W^T x$$

(We can re-write this as an inner product for succinctness.) There is another term, called the **bias**, that is just a constant factor.

$$z = \sum_{i=1}^{n} W_i x_i + b = W^T x + b$$

For mathematical convenience, we can actually incorporate it into our weight vector as W_0 and set $x_0 = +1$ for all of our inputs. (This concept of incorporating the bias into the weight vector will become clearer when we write code.)

$$z = \sum_{i=0}^{n} W_i x_i = W^T x$$

After taking the weighted sum, we apply an activation function, σ, to this and produce an activation a. The activation function for perceptrons is sometimes called a **step function** because, if we were to plot it, it would look like a stair.

$$\sigma(q) = \begin{cases} 1 & q \geq 0 \\ 0 & q < 0 \end{cases}$$

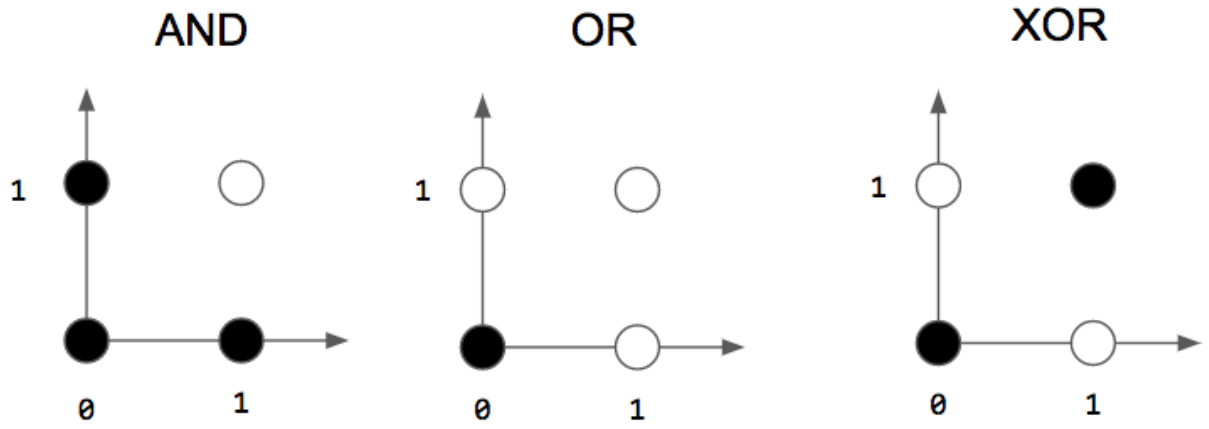
In other words, if the input is greater than or equal to 0, then we produce an output of 1. Otherwise, we produce an output of 0. This is the mathematical model for a single neuron, the most fundamental unit for a neural networks.

$$a = \sigma(W^T x)$$

Let's compare this model to the biological neuron. The inputs are analogous to the dendrites, and the weights model the synapse. We combine the weighted inputs by summing and send that weighted sum to the activation function. This acts as our all-or-nothing response function where 0 means the neuron didn't produce an output. Also note that our inputs and outputs are also binary, which is in accordance with the biological model.

Capabilities and Limitations of Perceptrons

Since the output of a perceptron is binary, we can use it for binary classification, i.e., an input belongs to only one of two classes. The classic examples used to explain what perceptrons can model are logic gates!



Let's consider the logic gates in the figure above. A white circle means an output of 1 and a black circle means an output of 0, and the axes indicate inputs. For example, when we input 1 and 1 to an AND gate, the output is 1, the white circle. We can create perceptrons that act like gates: they take 2 binary inputs and produce a single binary output!

However, perceptrons are limited to solving problems that are **linearly separable**. If two classes are linearly separable, this means that we can draw a single line to separate the two classes. We can do this easily for the AND and OR gates, but there is no single line that can separate the classes for the XOR gate! This means that we can't use our single-layer perceptron to model an XOR gate.

An intuitive way to understand why perceptrons can only model linearly separable problems is to look the weighted sum equation (with the bias).

$$\sum_{i=0}^{N} W_i x_i + b$$

This looks very similar to the equation of a line! (Or, more generally, a hyperplane.) Hence, we're creating a line and saying that everything on one side of the line belongs to one class and everything on the other side belongs to the other class. This line is called the **decision boundary**, and, when we use a single-layer perceptron, we can only produce one decision boundary.

In light of this new information, it doesn't seem like perceptrons are useful! But, in practice, many problems are actually linearly separable. Hope is not lost for non-linearly separably problems however! It can be shown that organizing multiple perceptrons into layers and using an intermediate layer, or **hidden layer**, can solve the XOR problem! This is the foundation of modern neural networks!

Single-Layer Perceptron Code

Now that we have a good understanding of how perceptrons works, let's take one more step and solidify the math into code. We'll use object-oriented principles and create a class. In order to construct our perceptron, we need to know how many inputs there are to create our weight vector. The reason we add one to the input size is to include the bias in the weight vector.

```
import numpy as np

class Perceptron(object):
    """Implements a perceptron network"""
    def __init__(self, input_size):
        self.W = np.zeros(input_size+1)
```

We'll also need to implement our activation function. We can simply return 1 if the input is greater than or equal to 0 and 0 otherwise.

```
def activation_fn(self, x):
    return 1 if x >= 0 else 0
```

Finally, we need a function to run an input through the perceptron and return an output. Conventionally, this is called the prediction. We add the bias into the input vector. Then we can simply compute the inner product and apply the activation function.

```
def predict(self, x):
    x = np.insert(x, 0, 1)
    z = self.W.T.dot(x)
    a = self.activation_fn(z)
    return a
```

All of these are functions of the Perceptron class that we'll use for perceptron learning.

Perceptron Learning Algorithm

We've defined a perceptron, but how do perceptrons learn? Rosenblatt, the creator of the perceptron, also had some thoughts on how to *train* neurons based on his intuition about biological neurons. Rosenblatt intuited a simple learning algorithm. His idea was to run each example input through the perceptron and, if the perceptron fires when it shouldn't have, inhibit it. If the perceptron doesn't fire when it should have, excite it.

How do we inhibit or excite? We change the weight vector (and bias)! The weight vector is a **parameter** to the perceptron: we need to keep changing it until we can correctly classify each of our inputs. With this intuition in mind, we need to write an update rule for our weight vector so that we can appropriately change it:

$$w \leftarrow w + \Delta w$$

We have to determine a good Δw that does what we want. First, we can define the error as the difference between the desired output d and the predicted output y.

$$e = d - y$$

Notice that when d and y are the same (both are 0 or both are 1), we get 0! When they are different, (0 and 1 or 1 and 0), we can get either 1 or -1. This directly corresponds to exciting and inhibiting our perceptron! We multiply this with the input to tell our perceptron to change our weight vector in proportion to our input.

$$w \leftarrow w + \eta \cdot e \cdot x$$

There is a **hyperparameter** η that is called the **learning rate**. It is just a scaling factor that determines how large the weight vector updates should be. This is a **hyperparameter** because it is not learned by the perceptron (notice there's no update rule for η!), but we select this parameter.

(For perceptrons, the Perceptron Convergence Theorem says that a perceptron will converge, given that the classes are linearly separable, regardless of the learning rate. But for other learning algorithms, this is a *critical* parameter!)

Let's take another look at this update rule. When the error is 0, i.e., the output is what we expect, then we don't change the weight vector at all. When the error is nonzero, we update the weight vector accordingly.

Perceptron Learning Algorithm Code

With the update rule in mind, we can create a function to keep applying this update rule until our perceptron can correctly classify all of our inputs. We need to keep iterating through our training data until this happens; one **epoch** is when our perceptron has seen all of the training data once. Usually, we run our learning algorithm for multiple epochs.

Before we code the learning algorithm, we need to make some changes to our init function to add the learning rate and number of epochs as inputs.

```
def __init__(self, input_size, lr=1, epochs=10):
    self.W = np.zeros(input_size+1)
    # add one for bias
    self.epochs = epochs
    self.lr = lr
```

Now we can create a function, given inputs and desired outputs, run our perceptron learning algorithm. We keep updating the weights for a number of epochs, and iterate through the entire training set. We insert the bias into the input when performing the weight update. Then we can create our prediction, compute our error, and perform our update rule.

```
def fit(self, X, d):
    for _ in range(self.epochs):
        for i in range(d.shape[0]):
            y = self.predict(X[i])
            e = d[i] - y
            self.W = self.W + self.lr * e * np.insert(X[i], 0, 1)
```

The entire code for our perceptron is shown below.

```
class Perceptron(object):
    """Implements a perceptron network"""
    def __init__(self, input_size, lr=1, epochs=100):
        self.W = np.zeros(input_size+1)
        # add one for bias
        self.epochs = epochs
        self.lr = lr

    def activation_fn(self, x):
```

```
        #return (x >= 0).astype(np.float32)
        return 1 if x >= 0 else 0

    def predict(self, x):
        z = self.W.T.dot(x)
        a = self.activation_fn(z)
        return a

    def fit(self, X, d):
        for _ in range(self.epochs):
            for i in range(d.shape[0]):
                x = np.insert(X[i], 0, 1)
                y = self.predict(x)
                e = d[i] - y
                self.W = self.W + self.lr * e * x
```

Now that we have our perceptron coded, we can try to give it some training data and see if it works! One easy set of data to give is the AND gate. Here's a set of inputs and outputs.

```
if __name__ == '__main__':
    X = np.array([
        [0, 0],
        [0, 1],
        [1, 0],
        [1, 1]
    ])
    d = np.array([0, 0, 0, 1])

    perceptron = Perceptron(input_size=2)
    perceptron.fit(X, d)
    print(perceptron.W)
```

In just a few lines, we can start using our perceptron! At the end, we print the weight vector. Using the AND gate data, we should get a weight vector of [-3, 2, 1]. This means that the bias is -3 and the weights are 2 and 1 for x_1 and x_2, respectively.

To verify this weight vector is correct, we can try going through a few examples. If both inputs are 0, then the pre-activation will be -3+0*2+0*1 = -3. When applying our activation function, we get 0, which is exactly 0 AND 0! We can try this for other gates as well. Note that this is not the *only* correct weight vector. Technically, if there exists a single weight vector that

can separate the classes, there exist an infinite number of weight vectors. Which weight vector we get depends on how we initialize the weight vector.

To summarize, perceptrons are the simplest kind of neural network: they take in an input, weight each input, take the sum of weighted inputs, and apply an activation function. Since they were modeled from biological neurons by Frank Rosenblatt, they take and produce only binary values. In other words, we can perform binary classification using perceptrons. One limitation of perceptrons is that they can only solve linearly separable problems. In the real world, however, many problems are actually linearly separable. For example, we can use a perceptron to mimic an AND or OR gate. However, since XOR is not linearly separable, we can't use single-layer perceptrons to create an XOR gate. The perceptron learning algorithm fits the intuition by Rosenblatt: inhibit if a neuron fires when it shouldn't have, and excite if a neuron does not fire when it should have. We can take that simple principle and create an update rule for our weights to give our perceptron the ability of learning.

Perceptrons are the foundation of neural networks so having a good understanding of them now will be beneficial when learning about deep neural networks!

Complete Guide to Deep Neural Networks Part 1

Neural networks have been around for decades, but recent success stems from our ability to successfully train them with many hidden layers. We'll be opening up the black-box that is deep neural networks and looking at several important algorithms necessary for understanding how they work. To solidify our understanding, we'll code a deep neural network from scratch and train it on a well-known dataset. Download the full code and dataset here.

Handwritten Digits: The MNIST Dataset

To motivate our discuss of neural networks, let's take a look at the problem of handwritten digit recognition.

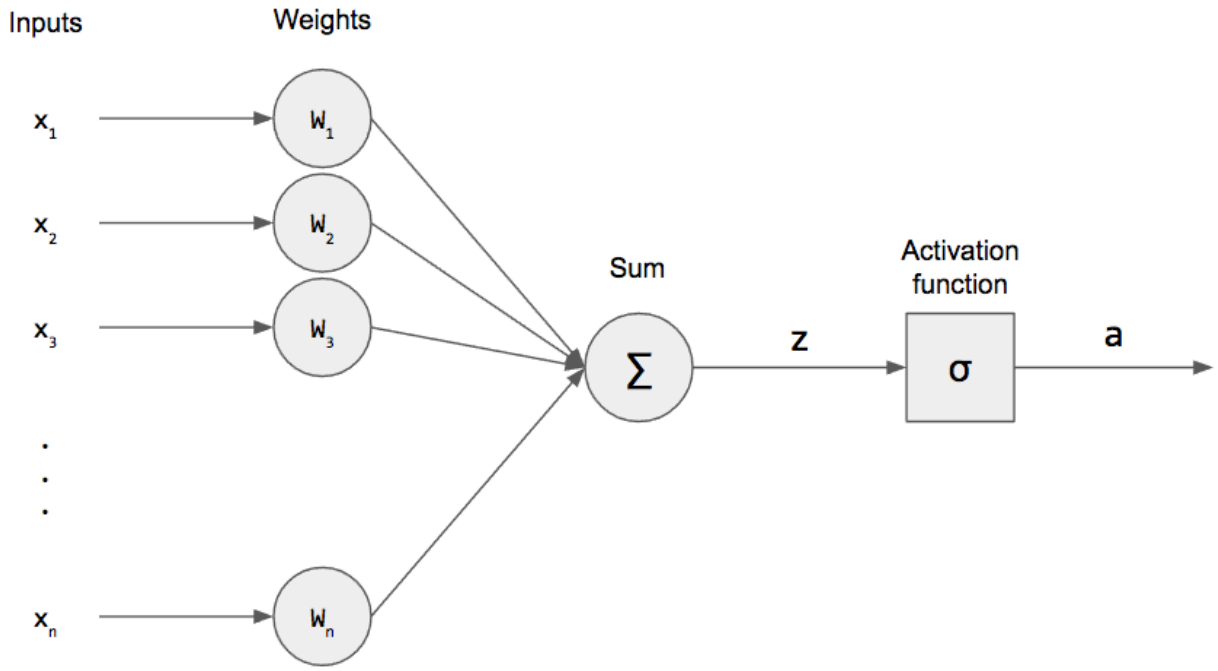
The goal is to determine the correct digit (0-9) given an input. In other words, we want to *classify* the images into 10 classes, one for each digit. This is more challenging than we may think: each new handwritten digit can have its own little variations, so using a fixed/static representation of a handwritten digit won't result in a good accuracy. However, machine learning is data-driven, and we can apply it to solve our problem. In particular, we'll be applying neural networks.

Luckily, we won't have to go out and collect this data ourselves. In fact, there's a very famous dataset, colloquially called MNIST, that we'll be using. In the earlier days, it was used for training the first modern convolutional neural network. It is still sometimes used as a dataset to train on and display results. In fact, there are still challenges to achieve the best accuracy! At the time of this writing, the state-of-the-art result on this dataset is 99.79% accuracy!

I've included the dataset in the ZIP file, and the above image is an example taken out of the dataset. To provide more information on the dataset, it consists of binary images of a single handwritten digit (0-9) of size 28×28. The provided **training set** (the data we use for training our network) has 60,000 images, and the **testing set** (the data we use for evaluating our network) has 10,000 images. We'll use it for our neural network and compare our results to the state-of-the-art.

Single-layer perceptrons recap

Before we start discuss multilayer perceptrons, if you're not already familiar with perceptrons, I highly recommending reading this post to acquaint yourself with these models since we will be starting from these small neural networks and adding more complexity.
To quickly recap single-layer perceptrons, a neuron had a graphical structure that looked like this.

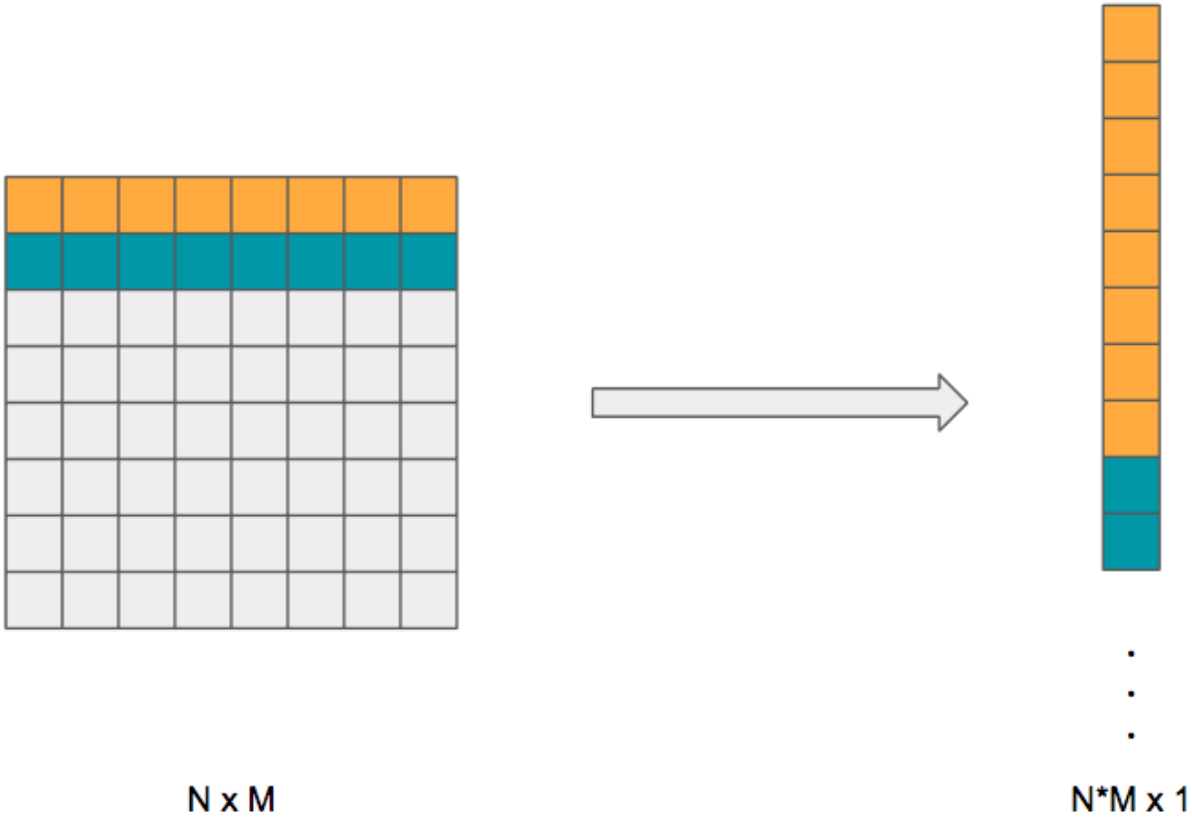


We take the weighted sum of our input (plus a bias term!) and apply a non-linear activation function. Mathematically, we can write the following statements.

$$z = \sum_{i=0}^{N} W_i x_i = \boldsymbol{W}^T \boldsymbol{x}$$

$$a = \sigma(z)$$

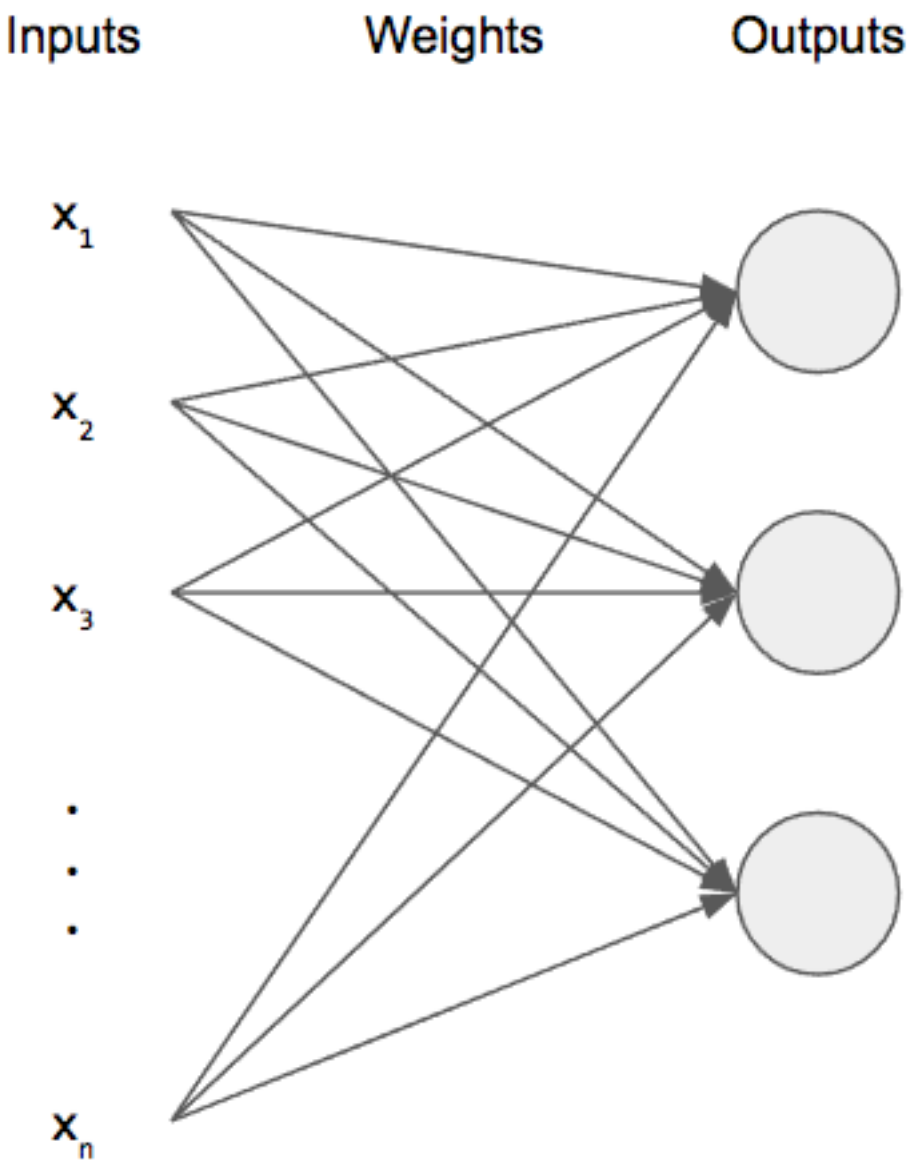
Remember that z is called the **pre-activation** and a is the **post-activation** because we applied the activation function. For our MNIST dataset, we have an image, not an input vector. So how can we convert a 2D image into a 1D vector? We simply flatten it!



We take the second row of our image and tack it on to the first; we tack on the third row to the concatenation of the first two and so on. What we end up with is a 1D vector representation of our 2D image. This is how we will feed in our inputs into our neural network. Specifically for our MNIST dataset, our images of 28×28 are flattened into a single $28 \cdot 28 \times 1 = 784 \times 1$ vector, which is the size of our input layer.
(The downside to this approach is that we lose spatial information. There is a type of neural network tailored for images called a **convolutional neural network** where we don't flatten the input image. These tend to do better at image tasks than regular neural networks.)

Multiple Output Neurons

Before we extend this to multiple layers, let's first extend this to multiple outputs. Right now, we only have a single, scalar output: a. While this is useful for binary classification, it isn't useful for much else. If we wanted to build a network that can classify more than two classes, we need to add more output neurons. For our MNIST dataset, since we have 10 classes (one for each digit), we'd need 10 output neurons. (For clarity of figure, I've only shown 3 output neurons).



Some of the mathematics changes when we do this: the **weight vector** becomes a **weight matrix** W with dimensions $3 \times N$, or, more generally, $\text{output layer size} \times \text{input layer size}$. Additionally, our pre- and post-activations become vectors z and a. The activation function σ is just applied element-wise to each component in vector z. The beauty of this generalization is that we can re-write our equations, and *they look almost identical* to when we had a single output neuron. We just now have vectors instead of scalars in some places and matrices instead of vectors in other places.

$$z = Wx + b$$

$$a = \sigma(z)$$

(I've included the bias explicitly and will continue to do so from here on.)

We can write this in terms of scalars as well. Suppose the input layer is indexed by j and output layer by k, we can re-write the above in component form.

$$z_k = \sum_j W_{kj} x_k + b_k$$

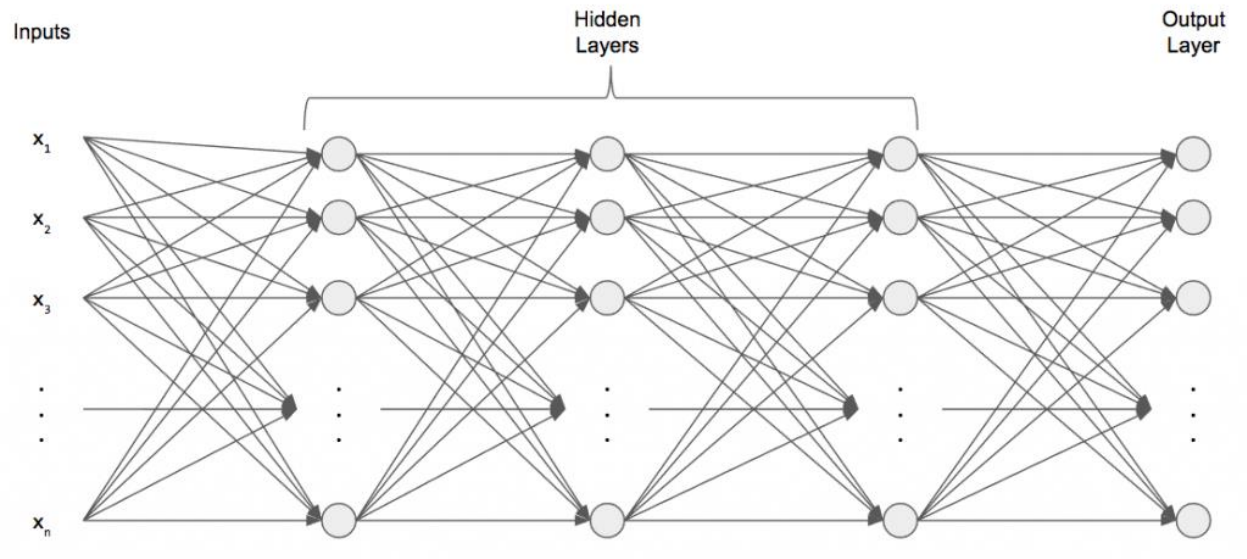
$$a_k = \sigma(z_k)$$

This is just saying that to get the pre-activation of an arbitrary neuron k in the output layer, we have to take each input, multiply it by the weight that connects that input to that kth neuron, and add the bias for that kth neuron. To convince yourself that this component-wise form is correct, see the above figure and only consider the first output neuron, i.e., $k = 1$. This component form will be useful later on when we discuss backpropagation: we'll start off by writing the component forms and then vectorize using linear algebra in the code. In general, vectorized code tends to run faster than non-vectorized code because of the myriad of libraries (e.g., numpy!) specializing in vectorized code.

But how do we structure our data (inputs and ground-truth outputs) for multi-class learning? The input doesn't change, but we take the ground-truth and encode it as a **one-hot vector**. We create a vector with the same length as the number of classes and put a single "1" in the position that corresponds to the correct class. Consider our MNIST dataset. If one of our inputs is actually a 4, then our ground-truth vector would be $[0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0]^T$. The vector is the zero vector, except we have a single "1" in the position of the correct class. (The first position corresponds to the digit zero.) Since the number of output neurons corresponds to the number of classes, the length of our vector is always the same as the number of output neurons.

Multilayer Perceptron Formulation

Now that we know how to account for multiple output neurons, we can finally get to the formulation for our **deep neural network**, or **multilayer perceptron**.



We still have an input and output layer, each with however many neurons we need. But in between those, we have any number of **hidden layers**, each with any number of neurons. These are called hidden layers because they are not directly connected to the outside world; they are hidden from the outside. Each neuron in a hidden layer is connected to each neuron in the next hidden layer. In the above figure, we have 3 hidden layers, so this is a 4-layer neural network. When we say the number of layers, we exclude the input layer because it really isn't a "layer" at all. This is also where the *deep* part of deep neural networks comes in: deep networks have many hidden layers!

So how do these deep neural networks function? They work in an iterative fashion: given an input, we perform the weighted sum (plus bias!) and apply the activation function. For the next hidden layer, we take the post-activations of the layer before it, hidden layer 1, and take the weighted sum (plus bias!) using a different weight matrix and apply the activation function. Then we repeat until we reach the output layer. Neural networks are also called **feedforward networks** for this exact reason: we feed our input forward through the network. The outputs of a hidden layer become the inputs to the next hidden layer.
Mathematically, we need to add another script for representing which layer we're referring to.

$$\boldsymbol{z}^{(l)} = \boldsymbol{W}^{(l)}\boldsymbol{a}^{(l-1)} + \boldsymbol{b}^{(l)}$$

$$\boldsymbol{a}^{(l)} = \sigma(\boldsymbol{z}^{(l)})$$

where $\boldsymbol{a}^{(1)} = \boldsymbol{x}$ and $l \in [2, L]$, i.e., there are L layers. We can, of course, write this is component form as well.

We're not taking exponents; l just means any layer. Also notice that in the pre-activation, we're no longer just referring to $\boldsymbol{x}$, but, rather, we take the activation from the previous layer $l-1$ when computing the pre-activatation of this layer l.

Training our Neural Network with Gradient Descent

Now that we've formulated and structured our neural network, let's see how we can train it. To train our neural network, we need some way of quantitatively measuring how well the network, with its current weights and biases, is performing. Just like with single-layer perceptrons, we introduce a **cost function** that measures the performance of our network. Here's an example of a cost function.

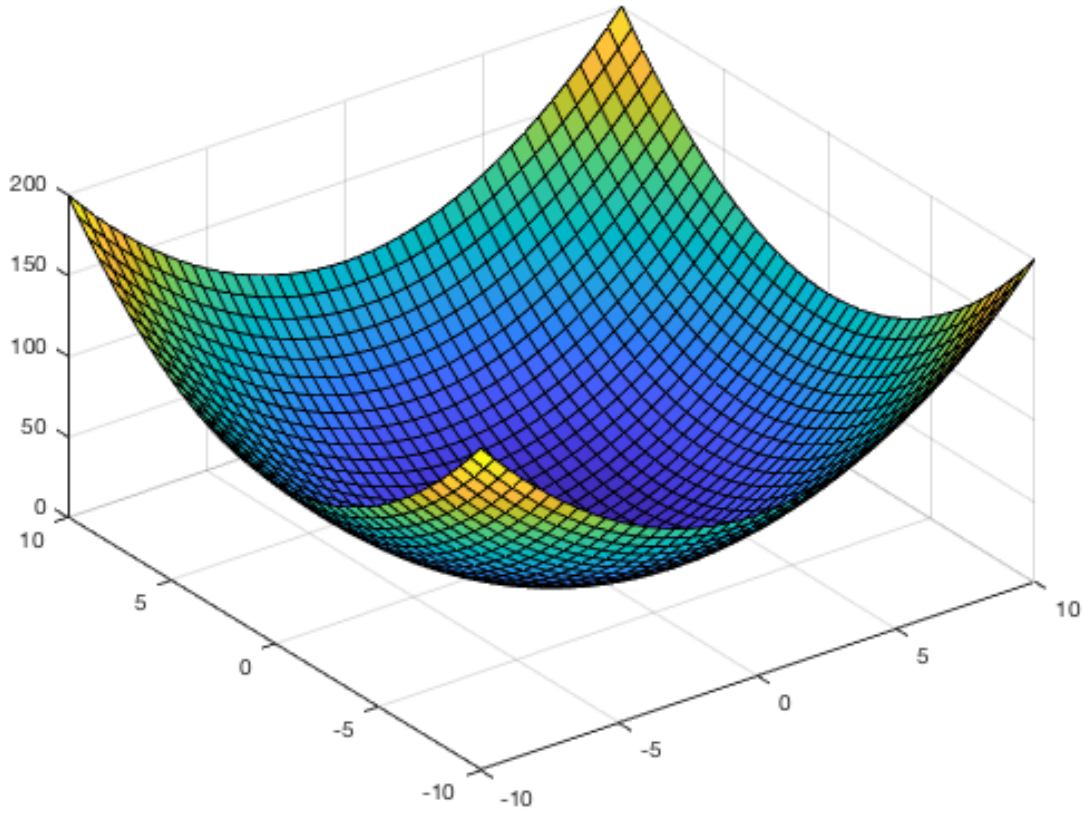
$$C(\boldsymbol{W}, \boldsymbol{b}) = \frac{1}{2} \sum_{\boldsymbol{x} \in \boldsymbol{X}} ||\boldsymbol{y}(\boldsymbol{x}) - \boldsymbol{a}^{(L)}(\boldsymbol{x})||^2$$

where $\boldsymbol{W}$ and $\boldsymbol{b}$ represent *all* of the weights and biases of our network, the sum is over all training examples, and $\boldsymbol{a}^{(L)}$ is the activation of the output layer given an input $\boldsymbol{x}$. $\boldsymbol{y}(\boldsymbol{x})$ represents the ground-truth label for input $\boldsymbol{x}$.

This is called the **quadratic cost** or **squared error** function. Intuitively, we take the squared difference between the network's answer and the ground-truth answer for each example. If both are vectors, we take the magnitude of the difference.

There are a few properties that all cost functions must follow that this one satisfies. First, it must be strictly greater than 0 everywhere *except at a single point*. That single point represents the minimum value of C and where the parameters are optimal: they give us a network that can correctly classify everything! Secondly, it must be smooth and differentiable: small differences between the predicted and ground-truth values should translate to a small value for the cost function. If we get a large cost value, that means our network isn't doing well; if we get a small cost value, then our network is great!

Why do we choose a quadratic cost? Why not a cubic or quartic or some other power? Quadratic functions look like parabolas and have a single global minimum. We can't say that of other powers (cubic, quartic, etc.). Using a quadratic-like function of two variables, we get a surface that looks kind of like this, where the x and y axes are parameters the z axis (upwards) is the value of the cost function.



Now that we have a way to quantify our network's performance with the cost function, we can apply the principle of gradient descent to train our network. For now, let's just consider a cost function with two variable $C(v_1, v_2)$ like the one shown above. We will later apply this princple to our neural network. Since the cost function tells us how well our network is doing, it makes sense to want to minimize this function cost. How do we minimize a function? Calculus! We could use calculus to find the partial derivatives and solve for the minimum. This will work for cost functions with a small number of variables, but, in most cases, neural networks have hundreds, thousands, or millions of parameters! So we have to think of something else.

There is a better way we can find the minimum value, and I'll explain it using an analogy. Imagine we're at a point on that quadratic surface in the above figure, and we're wearing a blindfold so we can't just see where the valley is and go right to there. How could we find the minimum? We could take a small step around where we are to find which direction the slope goes downhill. The we take a small step in that direction. Then, at that new point, we do the same thing: feel around for the direction that brings us downward and take a small step. We repeat this process until we reach the minimum.

We can solidify this analogy using calculus.

$$\Delta C = \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2$$

Intuitively, the change in the cost function is equal to how much we changed v_1 times how much the cost function changes with respect to v_1 plus how much we changed v_2 times how much the cost function changes with respect to v_2. The partial derivatives tell us how much changing one parameter affects the cost function. The goal is to change v_1 and v_2 so that ΔC is negative, therefore we keep moving towards the minimum cost.

In the analogy, we actually move to a new location on the surface, which called the **parameter space**. Going to that new position means we change our parameter values. This corresponds to updating the parameter values to reflect that small step. Like in our analogy, we want to take a small step downhill based on where we are locally. How do we know which direction is downhill in calculus? We can use the **gradient** to do this! The gradient is a vector that always points in the direction of increasing function value. We denote the gradient with an upside-down triangle ∇ called a **nabla**. More concretely, the gradient is the vector of partial derivatives of each of the parameters: $\nabla C = (\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2})^T$ since C is a function of two variables. This gradient is a vector that tells us where to go to *increase* the cost. Since we want to *decrease* the cost, we can move in the opposite direction of the gradient to get to a lower cost value.

$\Delta \boldsymbol{v} = (\Delta v_1, \Delta v_2)^T = -\eta \boldsymbol{\nabla C}$

η is called the **learning rate** and represents our step size. It is a **hyperparameter**, meaning that it isn't trained by our network: we choose it manually. We can re-write this as a parameter update rule.

$$v_1 \leftarrow v_1 - \eta \frac{\partial C}{\partial v_1}$$

$$v_2 \leftarrow v_2 - \eta \frac{\partial C}{\partial v_2}$$

$$\boldsymbol{v} \leftarrow \boldsymbol{v} - \eta \boldsymbol{\nabla C}$$

Our entire analogy is represented in those update equations above. Going back to neural networks, we can apply the same concept of gradient descent. In our case, we have weights and biases. We can write update rules for our weights and biases.

$$W_{jk}^{(l)} \leftarrow W_{jk}^{(l)} - \eta \frac{\partial C}{\partial W_{jk}^{(l)}}$$

$$b_j^{(l)} \leftarrow b_j^{(l)} - \eta \frac{\partial C}{\partial b_j^{(l)}}$$

Now we have equations that tell us how to update our neural network's parameters by going in the opposite direction of the gradient! *For each input*, we compute the gradient (namely $\frac{\partial C}{\partial W_{jk}^{(l)}}$ and $\frac{\partial C}{\partial b_j^{(l)}}$) and sum them all up. Only *then* can we apply these update rules!

We've discussed every term in our update rules except for the two key terms: $\frac{\partial C}{\partial W_{jk}^{(l)}}$ and $\frac{\partial C}{\partial b_j^{(l)}}$. How do we figure out what these are? To do this, we'll need to devise the backpropagation of errors algorithm, the *most important* algorithm for training deep neural networks. We use it to compute the partial derivative of the cost function *with respect to every parameter in every layer*. We'll delve into the details next time.

To recap, we learned about the handwritten digits dataset called MNIST (and many of our examples were tailored for that dataset to help solidify abstraction). We extended our perceptron from a single output to multiple output neurons by changing our weight vector to a weight matrix and our output scalar to an output vector. Then we defined the structure of multilayer perceptrons and some notation. Finally, we discussed the fundamental optimization algorithm for neural networks: gradient descent. We always step the parameters in the opposite direction of the gradient to minimize our cost function and train our neural network.
In the subsequent post, we'll see how to compute the gradient efficiently using the backpropagation of errors, or backprop, algorithm.

Complete Guide to Deep Neural Networks Part 2

Last time, we formulated our multilayer perceptron and discussed gradient descent, which told us to update our parameters in the opposite direction of the gradient. Now we're going to mention a few improvements on gradient descent and discuss the backpropagation algorithm that will compute the gradients of the cost function so we can update our parameters.

Improvements on Gradient Descent

Before moving on to backpropagation, let's discuss one point of practicality with gradient descent. Recall in vanilla gradient descent (also called **batch gradient descent**), we took each input in our training set, ran it through the network, computed the gradient, and summed all of the gradients for each input example. Only then did we apply the update rules. But if we have a large training set, we still have to wait to run *all* inputs before we can update our parameters, and this can take a long time!

Instead, we can take a randomly-sampled subset of our training data to represent the entire training set. For example, instead of running all 60,000 examples of MNIST through our network and then updating our parameters, we can randomly sample 100 examples and run that random sample through our network. That's a speedup factor of 600! However, we also want to make sure that we're using *all* of the training data. It wouldn't be a good use of the training set if we just randomly sampled 100 images each time. Instead, we randomly shuffle our data and divide it up into minibatches. At each minibatch, we average the gradient and update our parameters before moving on to the next minibatch. After we've exhausted all of our minibatches, we say one **epoch** has passed. Then we shuffle our training data and repeat!

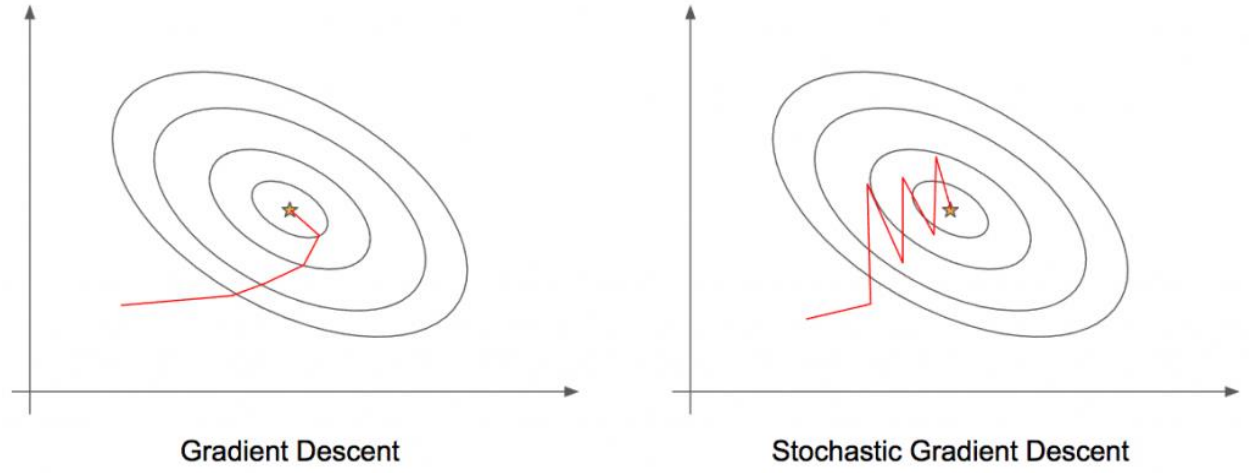
This technique is called **minibatch stochastic gradient descent**. Mathematically, we can write it like this.

$$W_{kj}^{(l)} \leftarrow W_{kj}^{(l)} - \eta \cdot \frac{1}{m} \sum_{\boldsymbol{x}_{(1)},\ldots,\boldsymbol{x}_{(m)}} \frac{\partial C_{\boldsymbol{x}}}{\partial W_{kj}^{(l)}}$$

$$b_{k}^{(l)} \leftarrow b_{k}^{(l)} - \eta \cdot \frac{1}{m} \sum_{\boldsymbol{x}_{(1)},\ldots,\boldsymbol{x}_{(m)}} \frac{\partial C_{\boldsymbol{x}}}{\partial b_{k}^{(l)}}$$

The sum is over all of the samples in our minibatch of size m. The batch size is another hyperparameter: we set it by hand. If we set it too large, then our network will take a longer time to train because it will be more like batch gradient descent. If we set it small, then the network

won't converge. Speaking of convergence, here's a visual representation of (batch) gradient descent versus (minibatch) stochastic gradient descent.



In this plot, the axes represent the parameters, and the ellipses are cost regions where the star is the minimum. Think of this as a 2D representation of a 3D cost function (see Part 1's surface plot). Each bend in the line represents a point in the parameter space, i.e., a set of parameters. The goal is to get to the star, the minimum. With gradient descent, the path we take to the minimum is fairly straightforward. However, with stochastic gradient descent, we take a much noisier path because we're taking minibatches, which cause some variation. That being said, stochastic gradient descent converges, i.e., reaches the minimum (or around there), much faster than gradient descent.

Here's an algorithm that describes minibatch stochastic gradient descent. We can repeat this algorithm for however many epochs we want.

1. Shuffle the training set and split into minibatches of size m. There will be $\lceil \frac{N}{m} \rceil$ minibatches of size m where $\lceil \cdot \rceil$ is the ceiling operator.
2. For each minibatch $\boldsymbol{x}_{(1)}, \ldots, \boldsymbol{x}_{(m)}$:
3. Run $\boldsymbol{x}_{(1)}, \ldots, \boldsymbol{x}_{(m)}$ through our network to and compute the average gradient, i.e.,

$$\frac{1}{m} \sum_{\boldsymbol{x}_{(1)}, \ldots, \boldsymbol{x}_{(m)}} \frac{\partial C_{\boldsymbol{x}}}{\partial W_{kj}^{(l)}}$$

$$\frac{1}{m} \sum_{\boldsymbol{x}_{(1)}, \ldots, \boldsymbol{x}_{(m)}} \frac{\partial C_{\boldsymbol{x}}}{\partial b_{k}^{(l)}}$$

4. and
5. Update the network's parameters

To recap, stochastic gradient descent drastically improves the speed of convergence by randomly selecting a batch to represent the entire training set instead of looking through the entire training set before updating the parameters.

Minibatch Stochastic Gradient Descent Code

Let's get to some code! First, we can start by creating a class for our multilayer perceptron. We'll need to pass in the number of layers and size of each layer as a list of integers. We also define the activation function to be a sigmoid function, but we'll discuss the details of the activation function a bit later. The exact activation function we choose is unimportant at this point since our algorithms will be generalizable to any activation function.

```
class MultilayerPerceptron(object):
    """Implementation of a MultilayerPerceptron (MLP)"""
    def __init__(self, layers):
        self.layers = layers
        self.num_layers = len(layers)
        self.activation_fn = sigmoid
        self.d_activation_fn = d_sigmoid

        self.weights = [np.random.randn(y, x) / np.sqrt(x) for x, y in
zip(layers[:-1], layers[1:])]
        self.biases = [np.random.randn(y, 1) for y in layers[1:]]
```

Then we initialize our weights and biases for each layer. Remember that each layer has a different number of neurons and thus requires a different weight matrix and bias vector for each.

One additional heuristic is to initialize the parameters to small, random numbers. Why don't we just initialize all of them to be the same number like one or zero? Think back to the structure of a neural network. To compute the input into the first hidden layer, we take the weighted sum of the weights and the input. If all of the weights were one, then *each hidden neuron receives the same value: the sum of all of the components of the input*. This defeats the purpose of having multiple neurons in the hidden layers! Instead, initialize to small, random numbers to break the symmetry. There are more sophisticated ways to initialize our weights and biases, but initializing them randomly works well.

Now we can define a few useful functions that can take an input X, where each row is a training vector, and feed it forward through network. In the case of MNIST and for the purpose of training, X is a matrix of size $m \times 784$ because our minibatch size is m and our flattened images are of size 784. The function is general enough to accept minibatches of any size. We iterate through all of the weights and biases of our network and perform $\sigma(\boldsymbol{W}\boldsymbol{a} + \boldsymbol{b})$. Note that

a starts off as x but is then updated in the innermost loop so that we're always referring to the activation of the previous layer.

```
def feedforward(self, X):
    activations = np.zeros((len(X), self.layers[-1]))
    for i, x in enumerate(X):
        a = x.reshape(-1, 1)
        for W, b in zip(self.weights, self.biases):
            a = self.activation_fn(W.dot(a) + b)
        activations[i] = a.reshape(-1)
    return activations

def accuracy(self, y, pred):
    return np.sum(np.argmax(y, axis=1) == np.argmax(pred, axis=1)) / len(y)
```

Additionally, we define a helper function that converts the one-hot vector into a number that we can use to compute the accuracy.

Now that we have those functions defined, let's get to implementing minibatch stochastic gradient descent. At the start of each epoch, we shuffle our training data and divide it up into minibatches. Then, for each minibatch except the last one, we call a function to update our parameters.

```
def fit(self, X, y, epochs=100, batch_size=100, lr=1e-4):
    N = len(y)
    for e in range(epochs):
        # shuffle and batch
        idx = np.random.permutation(N)
        shuffled_X, shuffled_y = X[idx], y[idx]
        batches = [zip(shuffled_X[b:b+batch_size],
shuffled_y[b:b+batch_size]) for b in range(0, N, batch_size)]

        for i in range(len(batches)-1):
            X_train_batch, y_train_batch = zip(*batches[i])
            self.update_params(X_train_batch, y_train_batch, lr)

        X_val_batch, y_val_batch = zip(*batches[-1])
        y_val_batch = np.array(y_val_batch)
        pred_val_batch = self.feedforward(X_val_batch)
        val_accuracy = self.accuracy(y_val_batch, pred_val_batch)
        print("Epoch {0}: Validation Accuracy: {1:.2f}".format(e+1,
val_accuracy))
```

Note that we update parameters for all of the batches *except one!* For this last batch, we compute the accuracy to act as a kind of "measure of progress" for our network. This is often called the **validation set.**

Let's take a look at how we update the parameters. nabla_W and nabla_b represent the total update for all of the weights and biases for this minibatch. In the loop, we accumulate the gradients so that we can apply the update rule in the last few lines of code. We average over the size of the training batch and multiply by the step size or **learning rate**.

```
def update_params(self, X_train_batch, y_train_batch, lr):
    nabla_W = [np.zeros_like(W) for W in self.weights]
    nabla_b = [np.zeros_like(b) for b in self.biases]

    for x, y in zip(X_train_batch, y_train_batch):
        delta_nabla_W, delta_nabla_b = self.backprop(x, y)
        nabla_W = [dnW + nW for nW, dnW in zip(delta_nabla_W, nabla_W)]
        nabla_b = [dnb + nb for nb, dnb in zip(delta_nabla_b, nabla_b)]

    self.weights = [W - (lr * nW / len(X_train_batch)) for W, nW in
zip(self.weights, nabla_W)]
    self.biases = [b - (lr * nb / len(X_train_batch)) for b, nb in
zip(self.biases, nabla_b)]
```

The most important step happens in the backprop function. This is where we compute the gradient for all of the weights and biases for an example.

Backpropagation

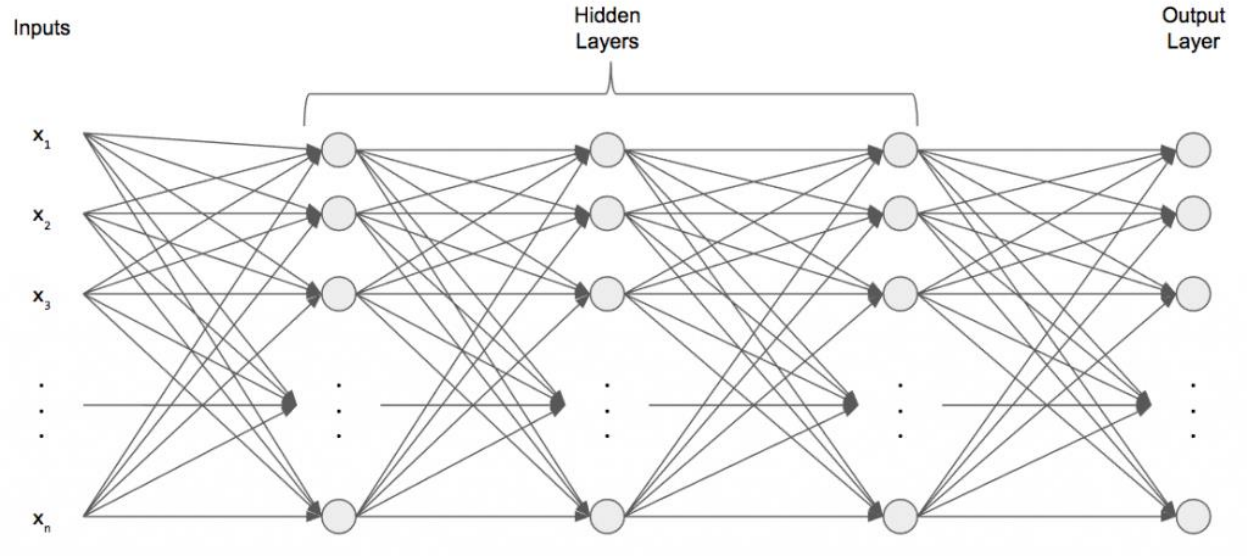
(The notation I use is consistent with Michael Nielsen in his book *Neural Networks and Deep Learning*.)

Now we can finally discuss backpropagation! It is the *fundamental and most important* algorithm for training deep neural networks. To start our discuss of backpropagation, remember these update equations.

$$W_{kj}^{(l)} \leftarrow W_{kj}^{(l)} - \eta \cdot \frac{1}{m} \sum_{\boldsymbol{x}_{(1)},\ldots,\boldsymbol{x}_{(m)}} \frac{\partial C_{\boldsymbol{x}}}{\partial W_{kj}^{(l)}}$$

$$b_{k}^{(l)} \leftarrow b_{k}^{(l)} - \eta \cdot \frac{1}{m} \sum_{\boldsymbol{x}_{(1)},\ldots,\boldsymbol{x}_{(m)}} \frac{\partial C_{\boldsymbol{x}}}{\partial b_{k}^{(l)}}$$

The ultimate goal is to solve for the partial derivatives of the cost function so we can update our parameters. Can't we just use calculus to solve for these? Well the answer isn't quite that simple. The problem is that each parameter isn't directly connected to the cost function. Consider a weight value connecting the input to the first hidden layer. It isn't directly connected to the cost function, but the value of the cost function *does* depend on that weight, i.e., it is a function of that weight (and every weight and bias in the network). So in order to update that weight in the earlier layers, we have to pass *backwards* through the later layers since the cost function is *closer* to the weights and biases towards the end. Hence the name **backpropagation**. Going backwards through the layers corresponds to using the **chain rule** from calculus, and we'll see more concrete examples of it.

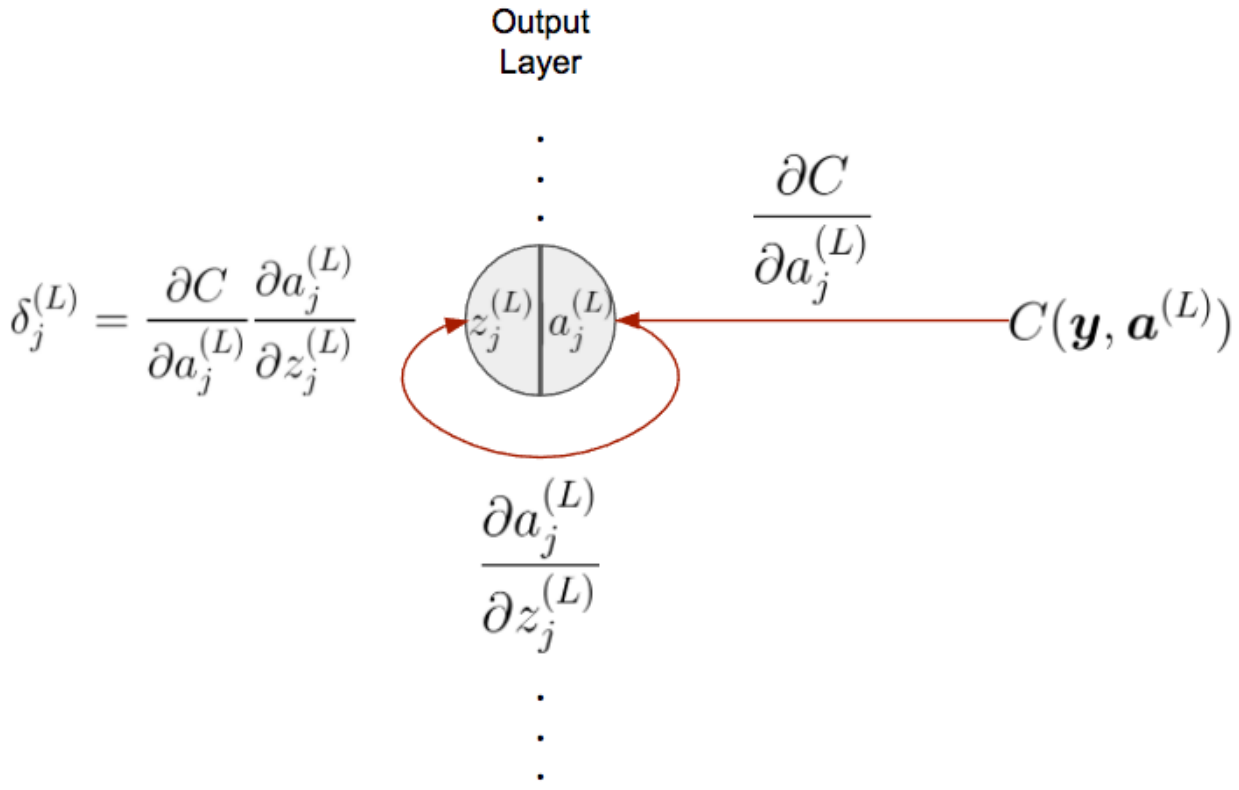


To make our lives easier, let's introduce an intermediary term called $\delta_j^{(l)}$ and define it like this.

$$\delta_j^{(l)} \equiv \frac{\partial C}{\partial z_j^{(l)}}$$

Intuitively, this represents how much changing a particular pre-activation (as a result of a change in a weight or bias) affects the cost function. We'll call this the **local error gradient**.

Now the only measure of quality of our network is at the very end: the cost function. So to start backpropagation, we need to figure out $\delta_j^{(L)}$ (suppose the output neurons are indexed by j). Consider any neuron j in the output layer. The cost function is directly a function of that neuron. We need to go backwards through our network, and that corresponds to *taking a derivative*! So to go back *through* the cost function to a particular neuron j, we take the derivative $\frac{\partial C}{\partial a_j^{(L)}}$. But this lands us at the post-activation $a_j^{(L)}$, and we need to get to the pre-activation. So we take another derivative to go from post-activation to pre-activation: $\frac{\partial a_j^{(L)}}{\partial z_j^{(L)}}$. Now we can compute $\delta_j^{(L)}$ by multiplying these derivatives together! In calculus, this is called the **chain rule**, and it is essential to backpropagation.



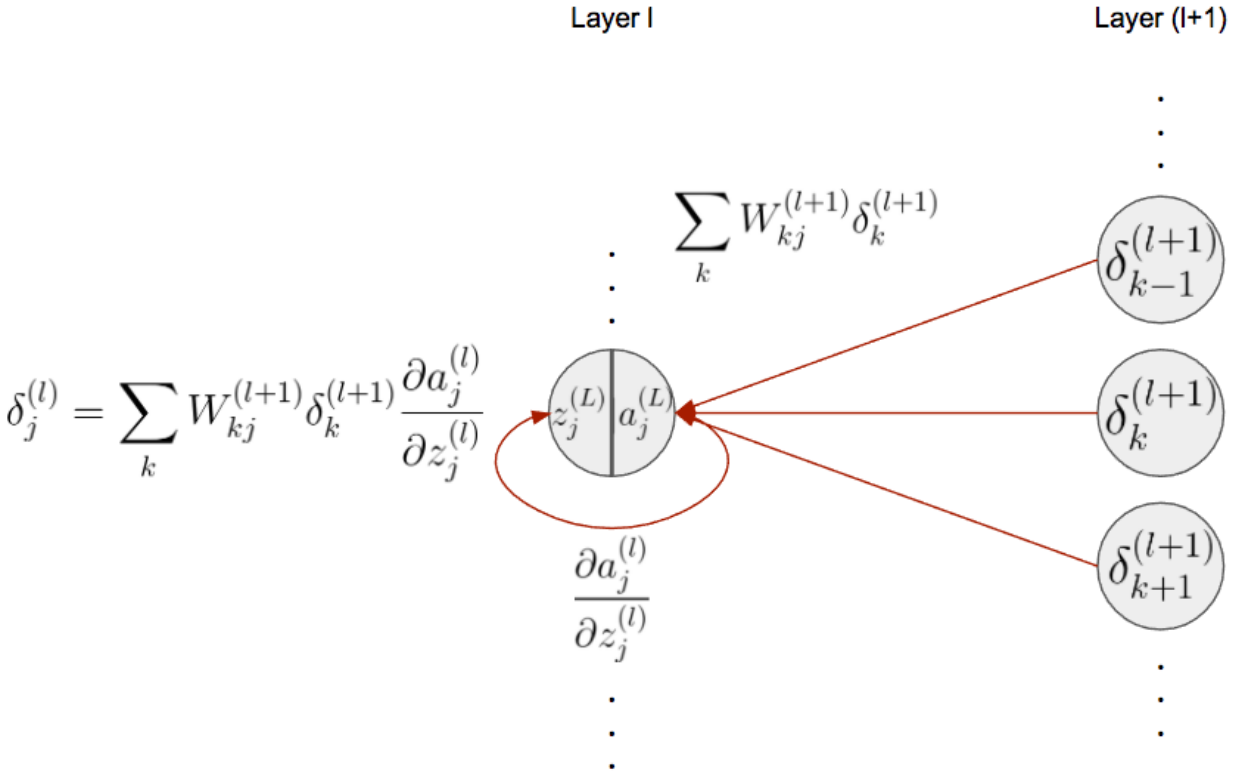
When we solve equation, we get the following.

$$\delta_j^{(L)} = \frac{\partial C}{\partial a_j^{(L)}} \sigma'(z_j^{(L)})$$

(1)

This is the first equation of backpropagation: computing the local error gradient at the last layer! Now that we have the last layer's local error gradient, we need an equation that tells us how to *propagate that error gradient backwards*.

Consider a neuron j in hidden layer l and all of the neurons, indexed by k in hidden layer $l+1$. We don't have to consider the $l+1$th layer's pre or post-activations since we've already backpropagated through them to get the all of the $\delta_k^{(l+1)}$ so we can use those as the starting point. To backprop through the weight matrix, we flip what index we sum over. Since we're going *to* a layer indexed by j from a layer indexed by k, we flip the index in the summation to sum over all of the neurons in layer $l+1$. Looking at the diagram, this makes sense because neuron j is connected to all neurons in the $l+1$th layer. After doing that, we land at the post-activation of layer l. Almost there! We can use the same trick in the previous equation with the partial derivative.



After expanding this equation, we get the following.

$$delta_j^{(l)} = \sum_k W_{kj}^{(l+1)} \delta_k^{(l+1)} \sigma'(z_j^{(l)})$$

Excellent! We're almost there! Now that we have a way to backpropagate our errors, we can use this local error gradient to calculate the partial derivatives of the weights and biases at a

given layer. This is the entire point of backpropagation! Unfortunately, there's no clever graphic we can refer to; we can directly use the chain rule to figure out the partial derivatives.

$$\frac{\partial C}{\partial W_{jk}^{(l)}} = \delta_j^{(l)} a_k^{(l-1)}$$

$$\frac{\partial C}{\partial b_j^{(l)}} = \delta_j^{(l)}$$

We've finally accomplished our goal! Now we can compute the gradient of the cost function with respect to each weight and bias at each layer! Now let's write down backpropagation as an algorithm and convert the algorithm to code.

1. Feed the input $\boldsymbol{a}^{(1)} = \boldsymbol{x}$ through the network and compute each $\boldsymbol{z}^{(l)}$ and $\boldsymbol{a}^{(l)}$ for $l \in [2, L]$.
2. Compute the error at the output layer using $\delta_j^{(L)} = \frac{\partial C}{\partial a_j^{(L)}} \sigma'(z_j^{(L)})$.
3. For each $l \in [L-1, 2]$
4. Compute the local error gradient $\delta_j^{(l)} = \sum_k W_{kj}^{(l+1)} \delta_k^{(l+1)} \sigma'(z_j^{(l)})$.
5. Compute the weight gradient $\frac{\partial C}{\partial W_{jk}^{(l)}} = \delta_j^{(l)} a_k^{(l-1)}$ and bias gradient $\frac{\partial C}{\partial b_j^{(l)}} = \delta_j^{(l)}$.

After we computed the gradients, we can apply gradient descent to update our weights and biases.

Backpropagation Code

Let's see what backpropagation looks like in code. Note the first step is to feed forward an input; the reason we can't use the feedforward function is because we need to cache the pre and post-activations so we can use them in the backward pass. The rest of this function is solidifies the mathematics into numpy code. We first compute ∇C and $\delta^{(L)}$, then we iterate through our layers backwards and use the formulas to compute ∇W and ∇b for each weight and bias in our network.

```
def backprop(self, x, y):
    nabla_W = [np.zeros(W.shape) for W in self.weights]
    nabla_b = [np.zeros(b.shape) for b in self.biases]

    # forward pass
    a = x.reshape(-1, 1)
    activations = [a]
```

```
    zs = []
    for W, b in zip(self.weights, self.biases):
        z = W.dot(a) + b
        zs.append(z)
        a = self.activation_fn(z)
        activations.append(a)

    # backward pass
    nabla_C = d_cost(y.reshape(-1, 1), activations[-1])
    delta = np.multiply(nabla_C, self.d_activation_fn(zs[-1]))
    nabla_b[-1] = delta
    nabla_W[-1] = delta.dot(activations[-2].T)

    for l in range(2, self.num_layers):
        z = zs[-l]
        delta = np.multiply(self.weights[-l+1].T.dot(delta),
self.d_activation_fn(z))
        nabla_b[-l] = delta
        nabla_W[-l] = delta.dot(activations[-l-1].T)
    return (nabla_W, nabla_b)
```

Note that I've vectorized the backprop equations to take full advantage of numpy. One extra thing we need is the derivative of the cost function with respect to the activation. Using the same quadric cost function, we can take the derivative to simply get the difference of the activation and ground-truth value.

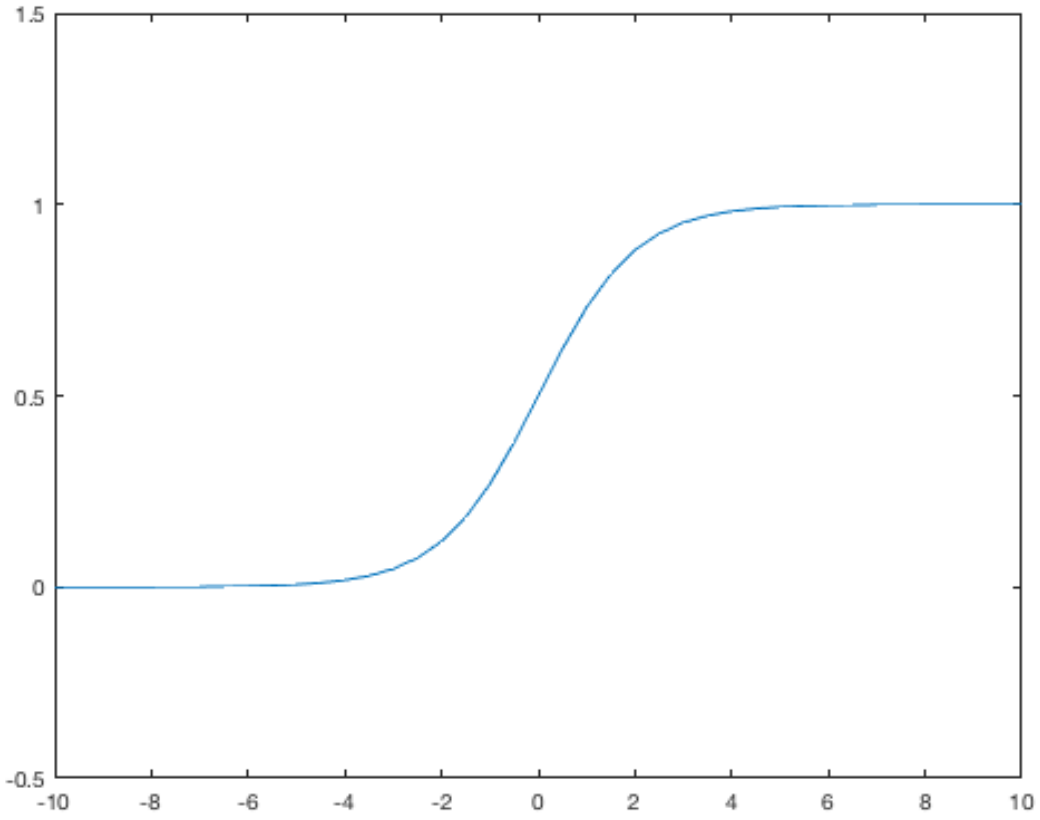
```
def d_cost(y, pred):
    return pred - y
```

Let's have a brief aside about the activation function. We've abstracted away the activation function, but, for backprop, we need to supply one and its derivative. We've seen the activation function for single-layer perceptrons, but it doesn't work too well for multilayer perceptrons. The reason is because of that pesky threshold. If we made a small change, we could get a completely different output! Instead, we want a smoother activation function: small changes to the inputs should result in small changes to the output so we know we're on the right track.

We can take that threshold function and smooth it out. The result is what we call a **logistic sigmoid**, or **sigmoid**, for short. The sigmoid is defined by this equation

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

If we plot it, it looks like this.



The sigmoid has several nice properties: it acts as a *squashing function* by taking its input and *squashing* it between zero and one; is smooth and differentiable; and a small Δz results in a small $\Delta\sigma(z)$.

In code, we can write the sigmoid and its derivative like this.

```
def sigmoid(z):
    return 1.0 / (1 + np.exp(-z))
def d_sigmoid(z):
    return sigmoid(z) * (1 - sigmoid(z))
```

In the past few years, sigmoids have fallen out of practice since they **saturate** for high and low inputs, meaning the derivative of the sigmoid is very small. Take a look at the ends of

the sigmoid: the slope hardly changes at all! Instead, researchers have favored a new activation function called a **Rectified Linear Unit** or ReLU for short.

We can use our new class very simply like this:

```
if __name__ == '__main__':
    X_train, y_train, X_test, y_test = load_mnist()

    # 2 layer network: 784 input layer, 512 hidden layer, and 10 output
layer
    mlp = MultilayerPerceptron(layers=[784, 512, 10])
    mlp.fit(X_train, y_train)

    pred = mlp.predict(X_test)
    accuracy = mlp.accuracy(y_test, pred)
    print("Test accuracy: {0:.2f}".format(accuracy))
```

The MNIST dataset and code to load MNIST is in the ZIP file!

Here's some output running for only 10 epochs with a batch size of 128.

```
Epoch 1: Validation Accuracy: 0.07
Epoch 2: Validation Accuracy: 0.08
Epoch 3: Validation Accuracy: 0.14
Epoch 4: Validation Accuracy: 0.14
Epoch 5: Validation Accuracy: 0.15
Epoch 6: Validation Accuracy: 0.19
Epoch 7: Validation Accuracy: 0.29
Epoch 8: Validation Accuracy: 0.30
Epoch 9: Validation Accuracy: 0.27
Epoch 10: Validation Accuracy: 0.30
Test accuracy: 0.32
```

To recap, we discussed an improvement on gradient descent called minibatch stochastic gradient descent. Instead of performing an update after seeing every example, we divide our training set into randomly sampled minibatches and update our parameters after computing the gradient averaged across that minibatch. We keep going until all of the minibatches have been seen by the network, then we shuffle our training set and start over! Next, we discussed the *fundamental and most important* algorithm for training deep neural networks: backpropagation!

Using this algorithm, we can compute the gradient of the cost function for each parameter in our network, regardless of the network's size!

Backpropagation and Gradient Descent are the two of the most important algorithms for training deep neural networks and performing parameter updates, respectively. A solid understanding of both are crucial to your deep learning success.

Introduction to Convolutional Networks for Vision Tasks

Neural networks have been used for a wide variety of tasks across different fields. But what about image-based tasks? We'd like to do everything we could with a regular neural network, but we want to explicitly treat the inputs as images. We'll discuss a special kind of neural network called a **Convolutional Neural Network (CNN)** that lies at the intersection between Computer Vision and Neural Networks. CNNs are used for a wide range of image-related tasks such as image classification, object detection/localization, image generation, visual question answering, and more! We'll discuss the different kinds of layers in a CNN and how they function. Finally, we will build a historical CNN architecture called LeNet-5 and use it to recognize handwritten digits using the Keras library (on top of Tensorflow).

If you're not familiar with neural networks or the MNIST dataset, I highly recommend acquainting yourself by reading the post here. You'll also need to install Tensorflow and Keras for your system.

Convolutional Neural Networks Overview

Let's look at the CNN we'll be constructing: LeNet-5.

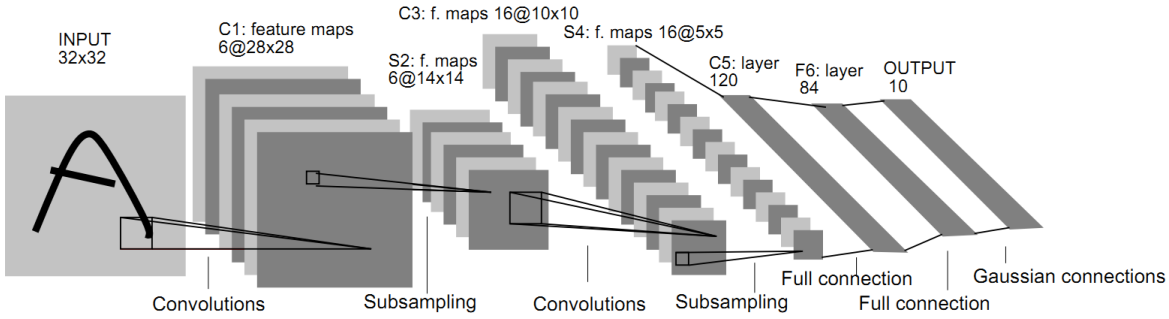
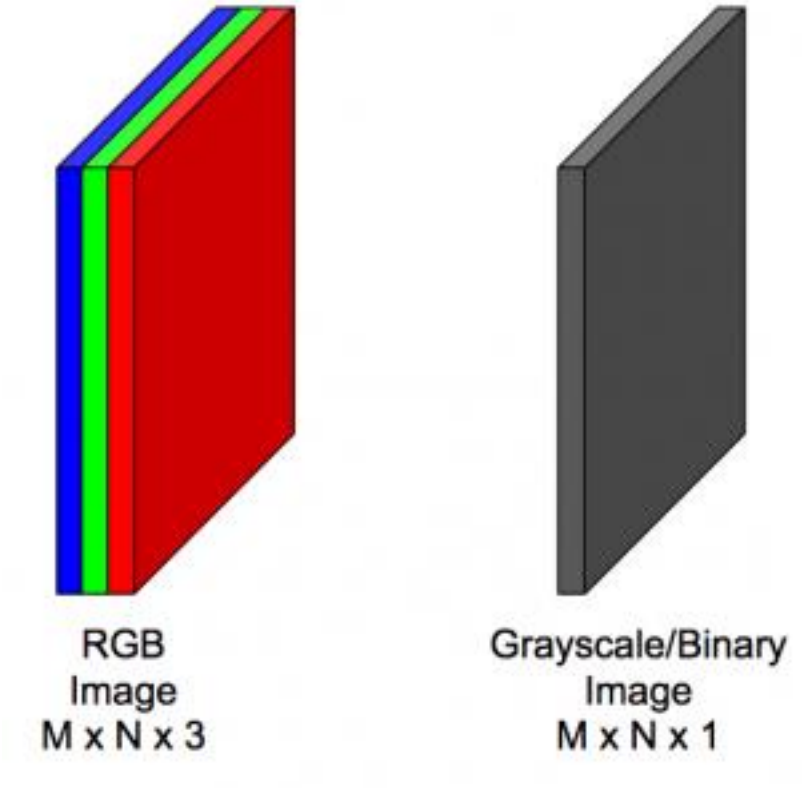


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

(*Gradient-Based Learning Applied to Document Recognition* by LeCun et al.)

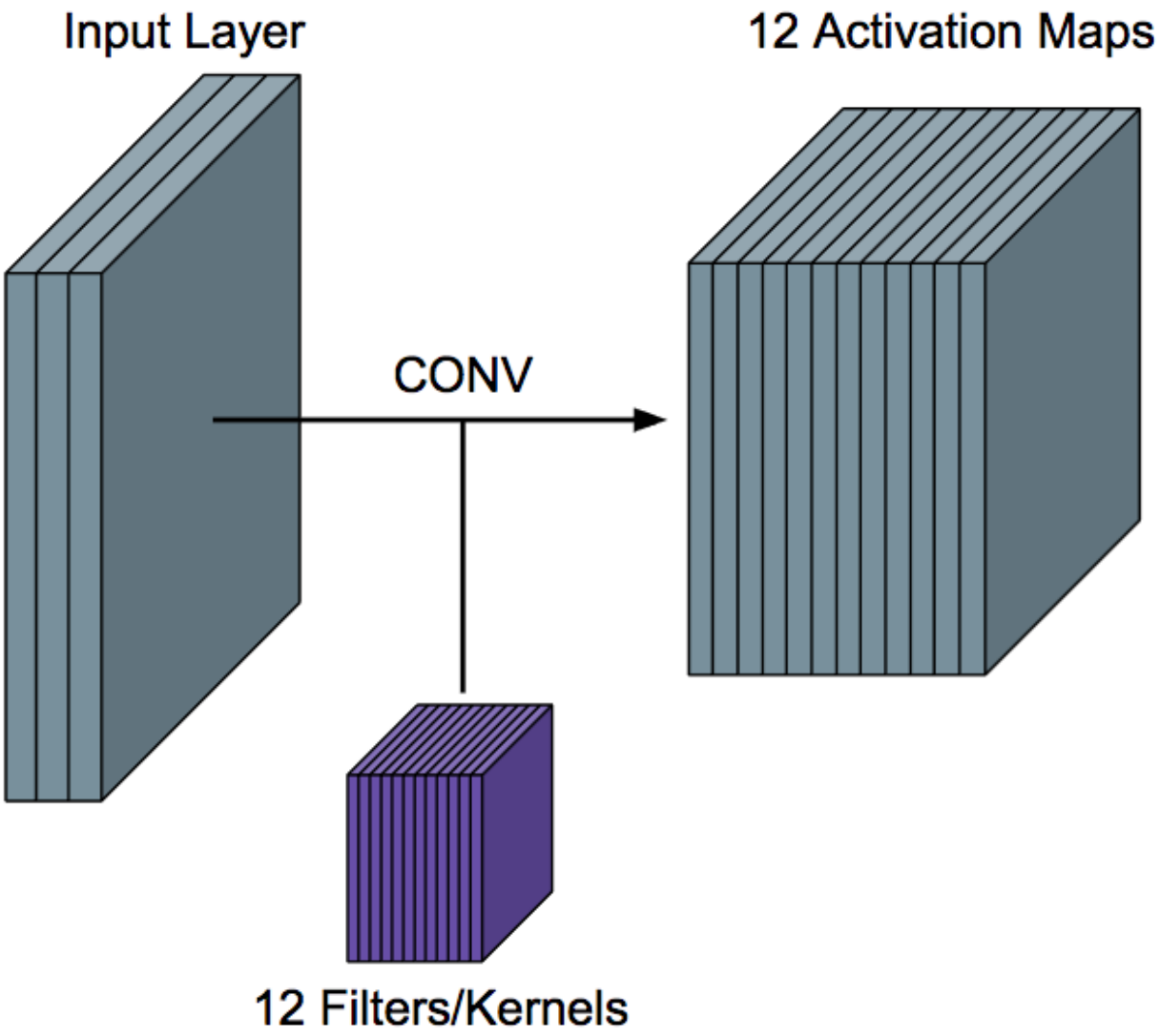
From this architecture, we notice is that the input is the full image itself. We don't flatten it in any way! This is good because we're maintaining the spatial relations between pixels. When we flatten it, we lose that spatial information.

Additionally, we have to think of the input in a different way: in terms of 3D **volumes**. This means that the image has a **depth** associated with it. This is also called the number of **channels**. For example, an RGB image of $M \times N$ has 3 channels (one for R, B, and G) so the full shape is really $M \times N \times 3$. On the other hand, a grayscale image of the same size only has one value per pixel location so its full shape is $M \times N \times 1$. Since these are technically 3-dimensionally, they are sometimes called **image volumes**.



In the LeNet architecture, we also notice that there seems to be three different kinds of layers: Convolutional Layers, Pooling Layers, and Fully-Connected Layers. These are the three fundamental kinds of layers that are present in all CNNs. New types of layers spring up all the time, but these three are always used! (Very recently, researchers are questioning the need for Pooling Layers instead of **Strided Convolutions**.)
Let's discuss each of these layers.

Convolution Layer

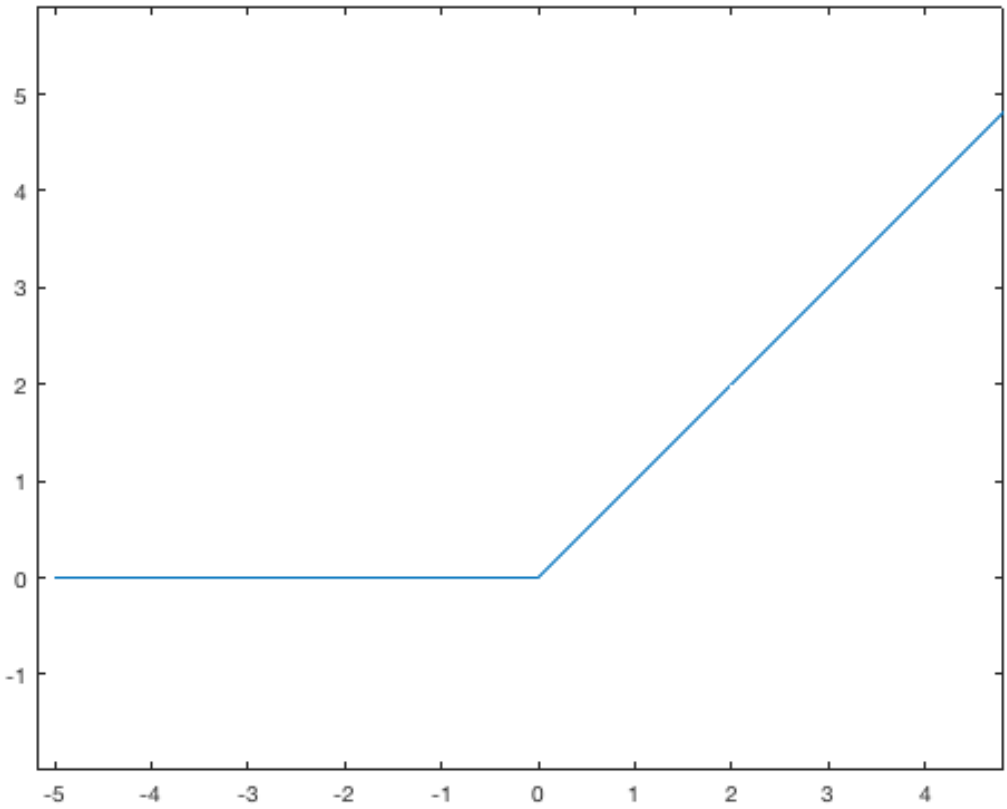


This is the most important layer in CNNs: it gives the CNN its name! The convolution layer, symbolized CONV, is where the feature learning happens. The idea is that we have a number of **filters or kernels**. These **filters** are just small patches that represent some kind of visual feature; these are the "weights" and "biases" of a CNN. We take each filter and **convolve** it over the input volume to get a single **activation map**. We won't discuss the details of convolution, but, intuitively, when we convolve a filter with the input volume, we get back an **activation map** that tells us "how well" parts of the input "responded" to the filter.

For example, if one of our filters was a horizontal line filter, then convolving it over the input would give us an activation map that indicates where the horizontal lines are in the input. The best part of CNNs is that these filters *are not hard-coded; they are learned!* We don't have to explicitly tell our CNN to look for horizontal lines; it will do so all by itself! During backprop, the CNN will learn that detecting features like horizontal lines will help it perform better so it will change the filters. This corresponds to updating the weights of an artificial neural network.
For a CONV layer, we have to specify at least the number of filters and their size (width and height). There are additional parameters we can specify as well, such as zero padding and stride, that we won't discuss. In terms of inputs and outputs, suppose a CONV layer receives an input of size $W_{\text{in}} \times H_{\text{in}} \times C_{\text{in}}$; assuming no zero padding and a stride of 1, we can compute the output width and height of the output activation maps to be $W_{\text{out}} = W_{\text{in}} - F_w + 1$ and $H_{\text{out}} = H_{\text{in}} - F_h + 1$, where F_w and F_h are the width and height of the filters. The output depth is simply the number of filters F! The output volume is hence $W_{\text{out}} \times H_{\text{out}} \times F$.

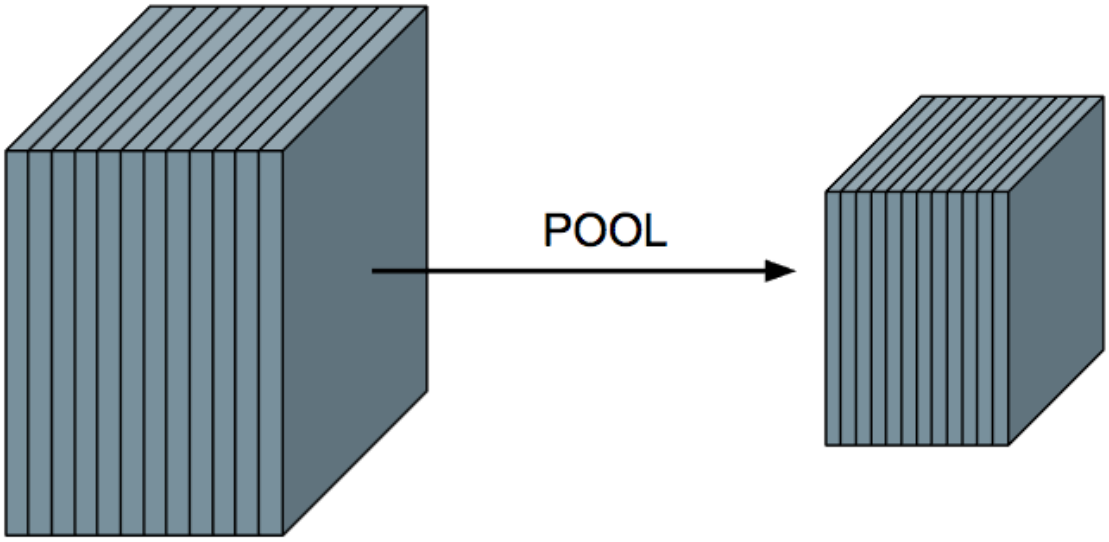
Let's do an example. In the first CONV layer of LeNet, we take a 32x32x1 input and produce an output of 28x28x6. Assuming no padding and a stride of 1, we can figure out that the 6 filters should each be of size 5x5. If we plug in 5 into the equations above, we should get 28: $32 - 5 + 1 = 28$.

Immediately following the CONV layer, just like an artificial neural network, we apply a non-linearity to each value in each activation map over the entire volume. Sigmoids aren't used that often in practice. The **Rectified Linear Unit or ReLU** is most frequently used for CNNs. The function is defined by the following equation.
$f(x) = \max(0, x)$



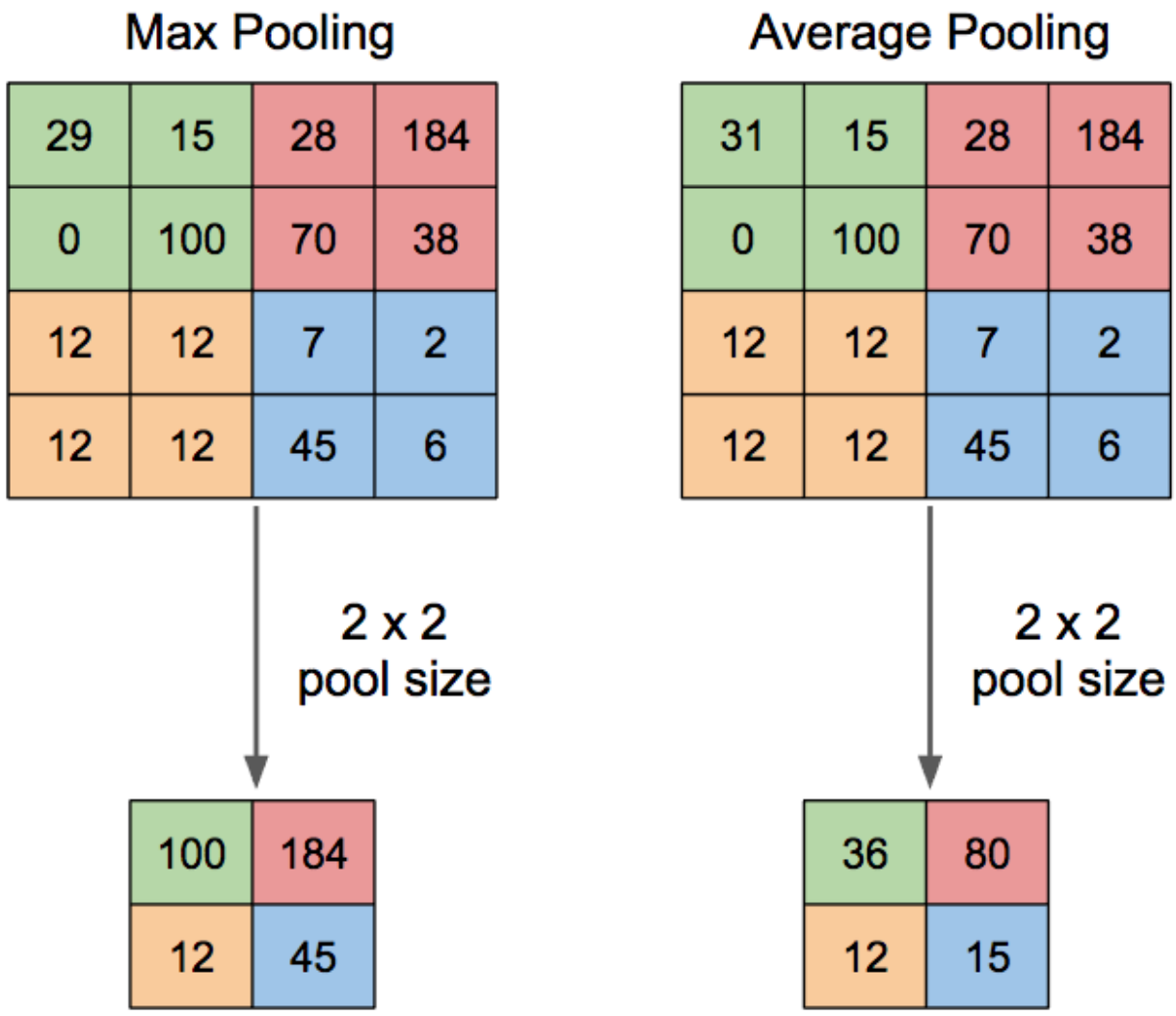
It is zero for $x \leq 0$ and x when $x > 0$. Unlike the sigmoid, the ReLU doesn't saturate (it has no upper limit). In practice, we find ReLUs work well.

Pooling Layer



This layer is primarily used to help reduce the computational complexity and extract prominent features. The pooling layer, symbolized as POOL, has no weights/parameters, unlike the CONV layers. The result is a smaller activation volume along the the width and height. The depth of the input is still maintained, e.g., if there are 12 activation maps that go into a POOL layer, the output will also have 12 activation maps.

For POOL layers, we have to define a pool size, which tells us by how much we should reduce the width and height of the activation volume. For example, if we wanted to halve the input activation volume along *both* width and height, we would choose a pool size of 2x2. If we wanted to reduce it by more, we choose a larger pool size. The most common pooling size is 2x2. The below figure shows how the pools are arranged. We take a little window the size of the pool, perform some computation, and then slide it over so it doesn't overlap and repeat.
The computation we do depends on the type of pooling: average or max.

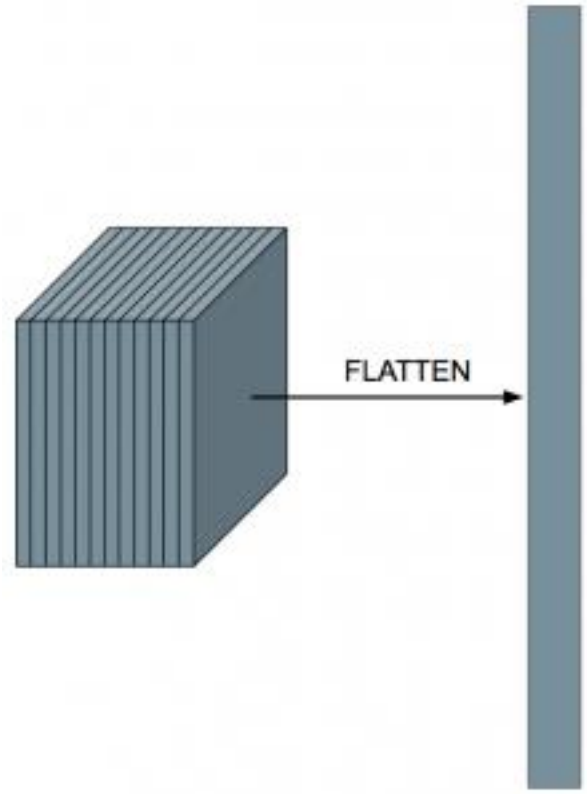


For max pooling, inside of the window, we just choose the maximum value in that window. This intuitively corresponds to choosing the most prominent features. For average pooling, we take the average of the values in the window. This produces smoother results than max pooling.

In practice, max pooling is used more frequently than average pooling, and the most common pooling size is 2x2.

Fully-Connected Layer

This layer is just a plain-old artificial neural network! The catch is that the input must be a vector. So if we have an activation *volume*, we must flatten it into a vector first!



After flattening the volume, we can treat this layer just like a neural network! It is okay to flatten here since we've already passed through all of the CONV layers and applied the filters.

Putting it all Together: The LeNet-5 Architecture

Now that we've discussed each of the layers independently, let's revisit the LeNet-5 architecture.

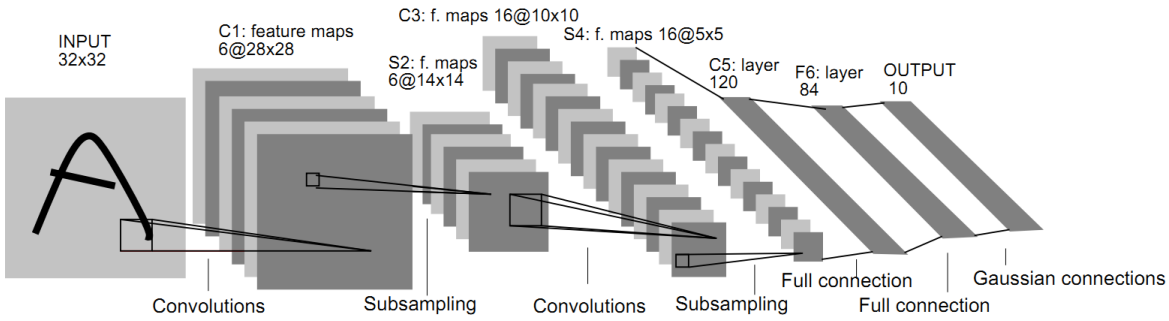


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

(*Gradient-Based Learning Applied to Document Recognition* by LeCun et al.)

The above figure isn't directly applicable to our case. The inputs we'll be dealing with are actually 28x28x1. But we can still perform a convolution so that the resulting volume is 28x28x6 by using some zero padding. We perform a convolution with 6 feature maps to get a resulting activation volume of 28x28x6. These filters will be 5x5. Then we subsample/pool to get 14x14x6 activation volume: we've halved the width and height so we use a pooling size of 2x2. Then we convolve again with 16 feature maps to get the activation volume of 10x10x16, meaning we must have used filters of size 5x5 (as an exercise, double-check this in the convolution equation). We pool again with a pooling size of 2x2. The last convolutional layer is 120 filters of size 5x5, which also flattens our activation volume into a vector. If it wasn't, then we would have to flatten it, which usually is the case. Finally, we add a hidden layer with 84 neurons and the output layer with 10 neurons.

The output layer neurons produce 10 numbers, but they don't really mean anything, so it would be better for our understanding to convert them into something interpretable like probabilities. We can use the **softmax** function to do this. Essentially, it takes the exponential of each of the 10 numbers and scales them by the sum of the exponentials of all of the numbers to make a valid probability distribution. Now we can interpret the output as a probability distribution! When we backprop, this softmax distribution is more useful than just a plain max function.

Now that we understand each of the pieces, let's start coding!

Coding LeNet-5 for Handwritten Digits Recognition

Since these layers are a bit more complicated than the a fully-connected layer, we won't be coding them from scratch. In real applications, we rarely do this for fundamental layers. We use a library to do this. Keras is a very simple, modular library that uses Tensorflow in the background and will help us build and train our network in clean and simple way.
First, we need to load the MNIST dataset. Keras can actually help us do this by calling a single function. If the dataset hasn't been downloaded yet, Keras will download it for us; if it has, Keras will load it and split it into a training and testing set.

```
import numpy as np
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D

BATCH_SIZE = 128
NUM_CLASSES = 10
NUM_EPOCHS = 20

(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

However, the data isn't the right shape for our CNN. If we look at the shape of the training inputs, we see that they have the shape $60,000 \times 28 \times 28$. The first dimension is the size of the training set, and the second and third are the image size. But we need to add another dimension to represent the number of channels. Numpy has a quick way we can do this.

```
X_train = X_train[..., np.newaxis]
X_test = X_test[..., np.newaxis]
```

Now our shape will be $60,000 \times 28 \times 28 \times 1$ for the training input and $10,000 \times 28 \times 28 \times 1$ for the testing input. This conforms with our CNN architecture! In general, the input shape usually has the batch size/number of examples, the image dimensions (width and height), and the number of channels. Be careful though, since sometimes the number of channels occurs before the image size (Theano does this) and sometimes it occurs after (Tensorflow does this).

There's some additional image processing cleanup we can do by rescaling so that each pixel value is $[0, 1]$.

```
X_train = X_train.astype('float32') / 255
X_test = X_test.astype('float32') / 255
```

As for the ground-truth labels, they're numbers, but we need them to be one-hot vectors since we're doing classification. Luckily, Keras has a function to automatically convert a list of numbers into a matrix of one-hot vectors.

```
y_train = keras.utils.to_categorical(y_train, NUM_CLASSES)
y_test = keras.utils.to_categorical(y_test, NUM_CLASSES)
```

Now its time to build our model! We'll be following as close to LeNet-5 as possible, but making a few alterations to improve learning. To build a model in Keras, we have to create a Sequential class and add layers to it.

```
model = Sequential()
model.add(Conv2D(6, (5, 5), activation='relu', padding='same',
input_shape=(28, 28, 1)))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(16, (5, 5), activation='relu'))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(120, (5, 5), activation='relu'))
model.add(Flatten())
model.add(Dense(84, activation='relu'))
model.add(Dense(NUM_CLASSES, activation='softmax'))
```

For the first layer, we have 6 filters of size 5x5 (with some zero padding) and a ReLU activation. (Keras also forces us to specify the dimensions of the input excluding the batch size). Then we perform max pooling using 2x2 regions. Then we repeat that combination except using 16 filters for the next block. Finally, we use 120 filters. In almost all cases, we would have to flatten our activation maps into a single vector. However, LeNet does not need to do this since at the last CONV layer, the size of the input activation map is already 5x5 so we would get a vector out anyways. Then we add an FC (or hidden) layer on top of that and finally the output layer.

We're finished! Now all that's left to do is "compile" our model and start training!

```
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])

model.fit(X_train, y_train,
          batch_size=BATCH_SIZE,
          epochs=NUM_EPOCHS,
          validation_split=0.10)
```

The categorical cross-entropy is a different loss function that works well for *categorical* data; we won't get to the exact formulation this time. As for the optimizer, we're using Adam (by Kingma and Ba) since it tends to converge better and quicker than gradient descent. Finally, we tell Keras to compute the accuracy. There are other metrics we could compute, but accuracy will suffice for our simple case (Keras will always compute the loss).

After our model has trained, we can evaluate it by running it on the testing data.

```
score = model.evaluate(X_test, y_test)
print('Test accuracy: {0:.4f}'.format(score[1]))
Expect to see a score in the high 90%'s! That's why CNNs are so awesome at
vision tasks!
During training Keras will helpfully display the progress like this.
Using TensorFlow backend.
Train on 54000 samples, validate on 6000 samples
Epoch 1/20
54000/54000 [==============================] - 9s - loss: 0.3757 - acc:
0.8864 - val_loss: 0.1183 - val_acc: 0.9662
Epoch 2/20
54000/54000 [==============================] - 9s - loss: 0.1049 - acc:
0.9676 - val_loss: 0.0779 - val_acc: 0.9770
Epoch 3/20
54000/54000 [==============================] - 9s - loss: 0.0709 - acc:
0.9782 - val_loss: 0.0675 - val_acc: 0.9807
Epoch 4/20
54000/54000 [==============================] - 9s - loss: 0.0554 - acc:
0.9824 - val_loss: 0.0597 - val_acc: 0.9830
Epoch 5/20
54000/54000 [==============================] - 9s - loss: 0.0437 - acc:
0.9863 - val_loss: 0.0429 - val_acc: 0.9877
Epoch 6/20
```

```
54000/54000 [==============================] - 9s - loss: 0.0364 - acc:
0.9889 - val_loss: 0.0407 - val_acc: 0.9877
Epoch 7/20
54000/54000 [==============================] - 9s - loss: 0.0302 - acc:
0.9904 - val_loss: 0.0481 - val_acc: 0.9868
Epoch 8/20
54000/54000 [==============================] - 9s - loss: 0.0276 - acc:
0.9911 - val_loss: 0.0399 - val_acc: 0.9885
Epoch 9/20
54000/54000 [==============================] - 9s - loss: 0.0236 - acc:
0.9919 - val_loss: 0.0406 - val_acc: 0.9880
Epoch 10/20
54000/54000 [==============================] - 9s - loss: 0.0197 - acc:
0.9938 - val_loss: 0.0419 - val_acc: 0.9872
Epoch 11/20
54000/54000 [==============================] - 9s - loss: 0.0189 - acc:
0.9936 - val_loss: 0.0368 - val_acc: 0.9895
Epoch 12/20
54000/54000 [==============================] - 9s - loss: 0.0151 - acc:
0.9948 - val_loss: 0.0438 - val_acc: 0.9873
Epoch 13/20
54000/54000 [==============================] - 9s - loss: 0.0154 - acc:
0.9949 - val_loss: 0.0468 - val_acc: 0.9870
Epoch 14/20
54000/54000 [==============================] - 9s - loss: 0.0121 - acc:
0.9960 - val_loss: 0.0411 - val_acc: 0.9892
Epoch 15/20
54000/54000 [==============================] - 9s - loss: 0.0123 - acc:
0.9955 - val_loss: 0.0447 - val_acc: 0.9877
Epoch 16/20
54000/54000 [==============================] - 9s - loss: 0.0097 - acc:
0.9968 - val_loss: 0.0538 - val_acc: 0.9867
Epoch 17/20
54000/54000 [==============================] - 9s - loss: 0.0101 - acc:
0.9965 - val_loss: 0.0457 - val_acc: 0.9887
Epoch 18/20
54000/54000 [==============================] - 9s - loss: 0.0080 - acc:
0.9974 - val_loss: 0.0473 - val_acc: 0.9893
Epoch 19/20
```

```
54000/54000 [==============================] - 9s - loss: 0.0105 - acc:
0.9964 - val_loss: 0.0540 - val_acc: 0.9858
Epoch 20/20
54000/54000 [==============================] - 9s - loss: 0.0087 - acc:
0.9972 - val_loss: 0.0482 - val_acc: 0.9890
 9760/10000 [============================>.] - ETA: 0s
Test accuracy: 0.9894
```

To recap, we discussed convolutional neural networks and their inner workings. First, we discussed why there was a need for a new type of neural network and why traditional artificial neural networks weren't right for the job. Then we discussed the different fundamental layers and their inputs and outputs. Finally, we use the Keras library to code the LeNet-5 architecture for handwritten digits recognition from the MNIST dataset.

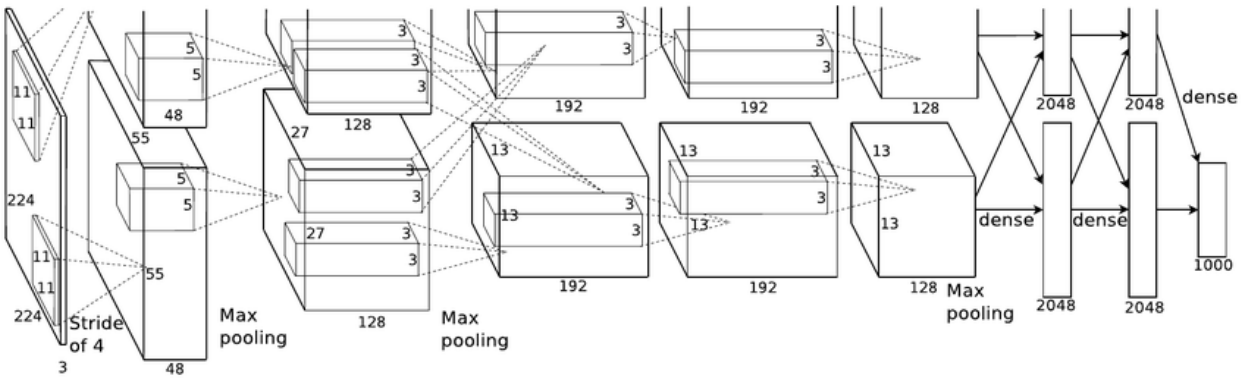
Convolutional Neural Networks are at the heart of all of the state-of-the-art vision challenges so having a good understand of CNNs goes a long way in the computer vision community.

Advanced Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are used in all of the state-of-the-art vision tasks such as image classification, object detection and localization, and segmentation. Previously, we've only discussed the LeNet-5 architecture, but that hasn't been used in practice for decades! We'll discuss some more modern and complicated architectures such as GoogLeNet, ResNet, and DenseNet. These are the state-of-the-art networks that won challenges such as the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), CIFAR-10, and CIFAR-1000.

AlexNet

The paper that started the entire deep learning craze: *ImageNet Classification with Deep Convolutional Neural Networks* by Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. AlexNet showed that we can efficiently train deep convolutional neural networks using GPUs.



Source (*ImageNet Classification with Deep Convolutional Neural Networks* by Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton) (Fun fact: the figure is taken from the actual paper and is actually cut off!)

AlexNet uses model parallelism to split the network across multiple GPUs. The architecture uses 4 groupings of convolutional, activation, and pooling layers, and 3 fully-connected layers of 4096, 4096, and 1000, where the output vector has a dimensionality of 1000.

However, we also use an additional layer called a **Batch Normalization** layer. The motivation behind it is purely statistical: it has been shown that normalized data, i.e., data with zero mean and unit variance, allows networks to converge much faster. So we want to normalize our mini-batch data, but, after applying a convolution, our data may not still have a zero mean and unit variance anymore. So we apply this batch normalization after each convolutional layer.
The implementation of Batch Normalization itself is pretty straightforward.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

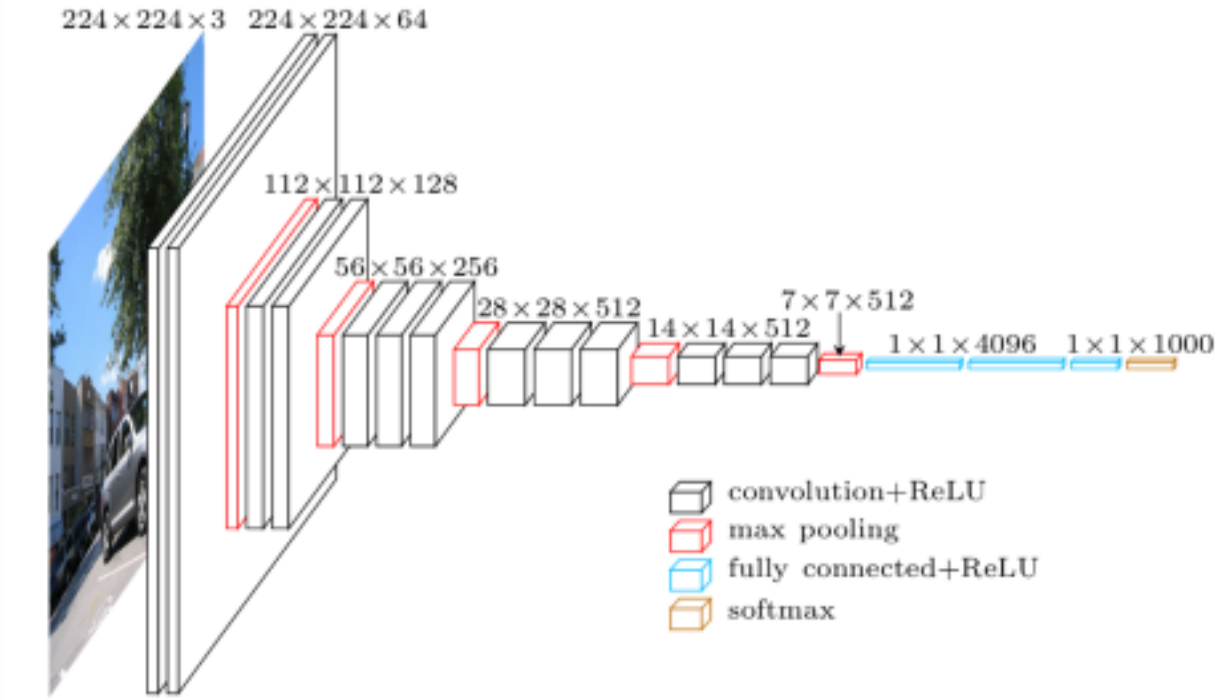
$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Source (*Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift* by Sergey Ioffe and Christian Szegedy)

There are two parameters to this batch normalization layer: γ and β. We compute two quantities: the mean over the batch and the variance over the batch. Each input x_i is actually a vector (technically, a tensor). Hence, the mean and variance are also tensors. After computing these quantities, we have a normalization step where we shift each point so it has a zero mean and unit variance. The ε is just a small constant added for numerical stability. Then, we apply two transforms: scaling by γ and shifting by β. These values are learned by the network during the backward pass. When introducing this new layer into our network, we have to backpropagate through it. Certainly batch normalization can be backpropagated over, and the exact gradient descent rules are defined in the paper.
Since we apply batch normalization after each convolution, our AlexNet architecture is now 4 groups of (Convolution, *Batch Normalization*, Activation Function, and Pooling). Then, we flatten the remaining image into a vector and pass it through 2 fully-connected layers of 4096 neurons each (they split this across two GPUs) and the final output layer of 1000 neurons.

VGG-Net

After AlexNet, the next innovation was VGG-Net, which came in two variants: VGG16 and VGG19. These networks had several improvements over AlexNet. First of all, this network was deeper than AlexNet. Work had been done just a year prior by Zeiler and Fergus that demonstrated deeper networks learned better. And, as we progress deeper into the network, the convolutional filters start to build on each other: the earlier layers are just line and edge detectors, but the later layers combine the earlier layers into shape and face detectors!



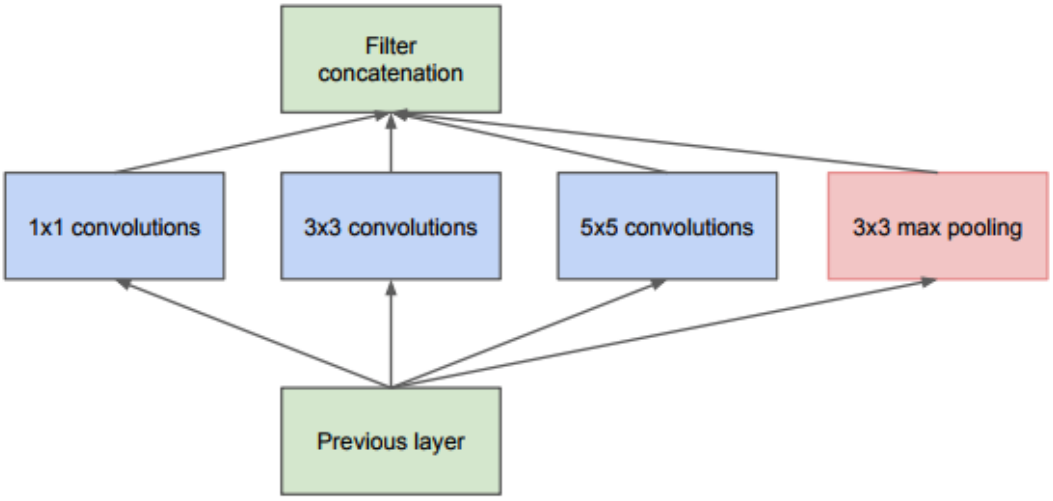
Source (https://www.cs.toronto.edu/~frossard/post/vgg16/)

Another improvement was that VGGNet used successive convolutional layers instead of the CONV - POOL architecture defined by LeNet. In the network, there are blocks of sequential convolutional layers, e.g., the CONV - CONV - CONV in the middle and end of the network. The danger of using successive convolutions at the time was that the network would become too large and training would take a long time to train. However, with new GPUs providing more computational power, this concern was quickly dismissed. This is the upwards trend on depth and computation that we'll notice as we progress.

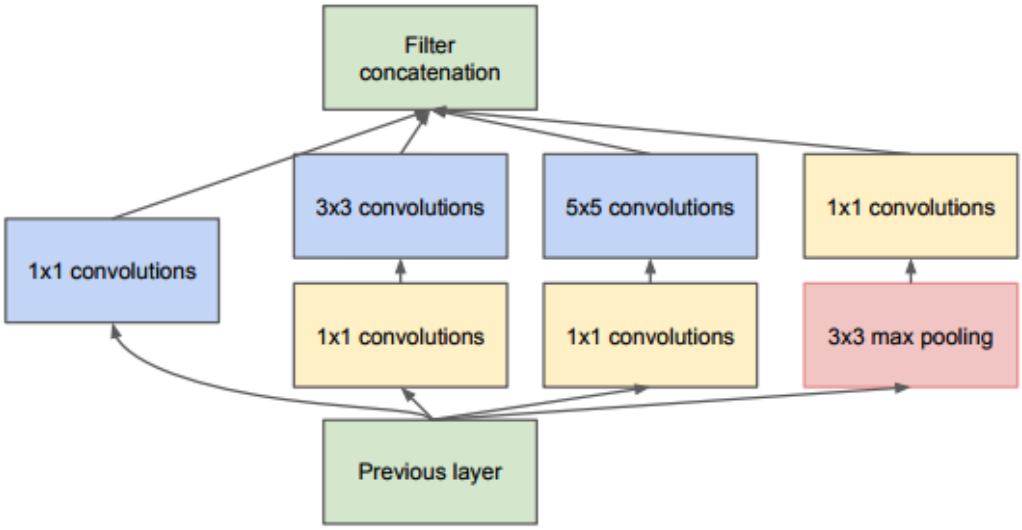
VGG-Net produced a smaller error than AlexNet in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2014.

GoogLeNet

The network that won the ILSVRC in 2014 was GoogLeNet from Google, of course. The real novelty of this network is the **Inception Modules**. The network itself simply stacked these inception modules together.



(a) Inception module, naïve version



(b) Inception module with dimensionality reduction

Figure 2: Inception module

Source (*Going Deeper with Convolutions* by Szegedy et al.)

The motivation behind these inception modules was around the issue of selecting the right filter or kernel size.We always prefer to use smaller filters, like 3x3 or 5x5 or 7x7, but which

ones of theses works the best? Depending on how deep we make our network, each convolutional layer has a choice between 3 different filter sizes. If our network was d layers deep, then we essentially have to try 3^d possible combinations, i.e., for each convolutional layer, we have 3 possible combinations so $3 \cdot 3 \cdot 3 \ldots 3\, d$ times is 3^d.

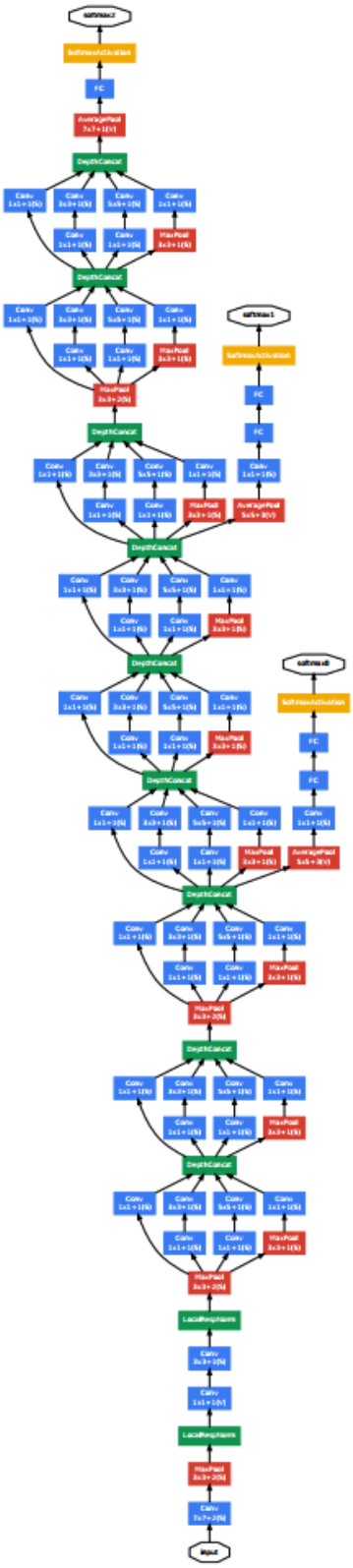
Instead of doing this, the inception modules just accept that fact that choosing this is difficult: so why not choose all of them? Instead of choosing just a single filter size, choose all of them and concatenate the results. Taking the activation maps of features from the previous layer, we apply 3 separate convolutions and one pooling on top of that single input and concatenate the resulting feature maps together.
For example, if our input was $h_1 \times w_1 \times d_1$ and our output feature maps were $h_2 \times w_2$, then we can apply convolutions to our input to produce $h_2 \times w_2 \times \sum_i d^{(i)}$ where the sum is over all of the number of filters for all of the convolutional layers. We have to adjust the padding and spacing of our convolutions since the pooling layer will reduce the width and height of our input and the outputs must all have to same width and height to concatenate together.

There is one issue with this approach: computational complexity. An inception module is slow for high-dimensionality input. The solution is to convert that high-dimensionality input to a lower dimensionality using 1x1 convolutions. *These kinds of convolutions are actually fully-connected layers!*

For example, if our input was $64 \times 64 \times 128$, applying a convolutional layer of 1x1 convolutions with 32 filters produces an output of $64 \times 64 \times 32$ (assuming no padding). This effectively reduced the dimension of our input! Those 32 1x1 filters are really just single neurons and their weights are learned just like everything else in the network!

The improved inception modules use these dimensionality-reducing 1x1 convolutions before applying the 3x3 and 5x5 convolutional layers. We also perform dimensionality reduction after the max pooling layer as well, for the same reason.

Now that we understand how an inception module works, we can apply them in GoogLeNet. As mentioned before, we simply stack many of these inception modules on top of each other to create GoogLeNet. (There are a few layers before we get to the stacks of inception module, and the graphic also shows auxiliary classifiers.)



Source (*Going Deeper with Convolutions* by Szegedy et al.)

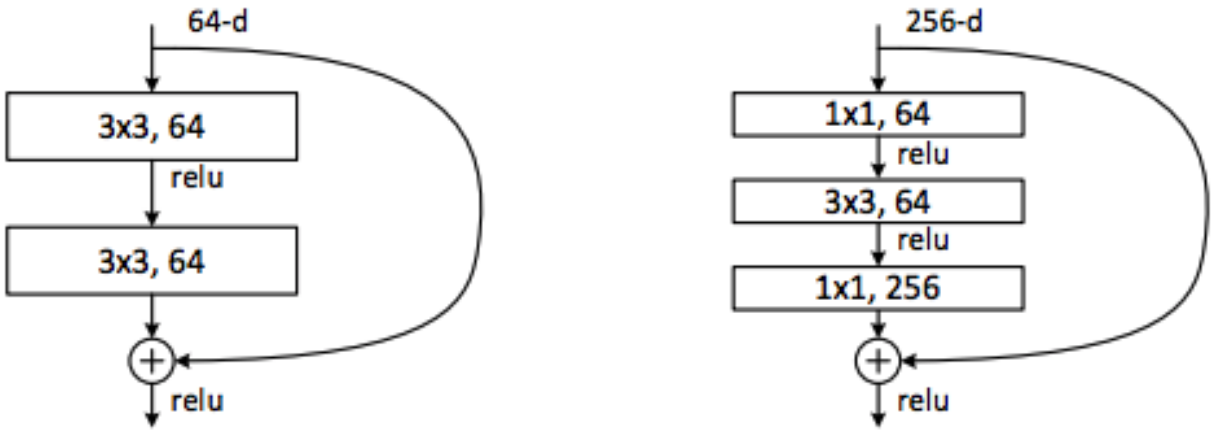
This network was *even deeper* than the VGG19 network that also participated in the ILSVRC the same year. GoogLeNet won the image classification challenge in 2014, and won again in 2016 after they iterated on it (Inception v4).

ResNet

The network that won the 2015 ILSVRC is also the deepest network to this day: ResNet, or residual networks. The major issue with taking something like GoogLeNet and extending it many layers is the **vanishing gradient problem**.

With any network, we compute the error gradient at the end of the network and use backpropagation to propagate our error gradient backwards through the network. Using the chain rule, we have to keep multiplying terms with the error gradient as we go backwards. However, in the long chain of multiplication, if we multiply many things together that are less than 1, then the final result will be very small. This applies to the gradient as well: the gradient becomes very small as we approach the earlier layers in a deep architecture. This small gradient is an issue because then we can't update the network parameters by a large enough amount and training is very slow. In some cases, the gradient actually becomes zero, meaning we don't update the earlier parameters at all!

Whenever we backpropagate through an operation, we have to use the chain rule and multiply, but what if we were to backpropagate through the identity function? Then the gradient would simply be multiplied by 1 and nothing would happen to it! This is the idea behind ResNet: it stacks these **residual blocks** together where we use an identity function to preserve the gradient.



Source (*Deep Residual Learning for Image Recognition* by He et al.)

The beauty of residual blocks is the simplicity behind it. We take our input, apply some function to it and add it to our original input. Then, when we take the gradient, it is simply 1.

Mathematically, we can represent the residual block like this.
$H(x) = F(x) + x$
So when we find the partial derivative of the cost function C with respect to x, we get the following.

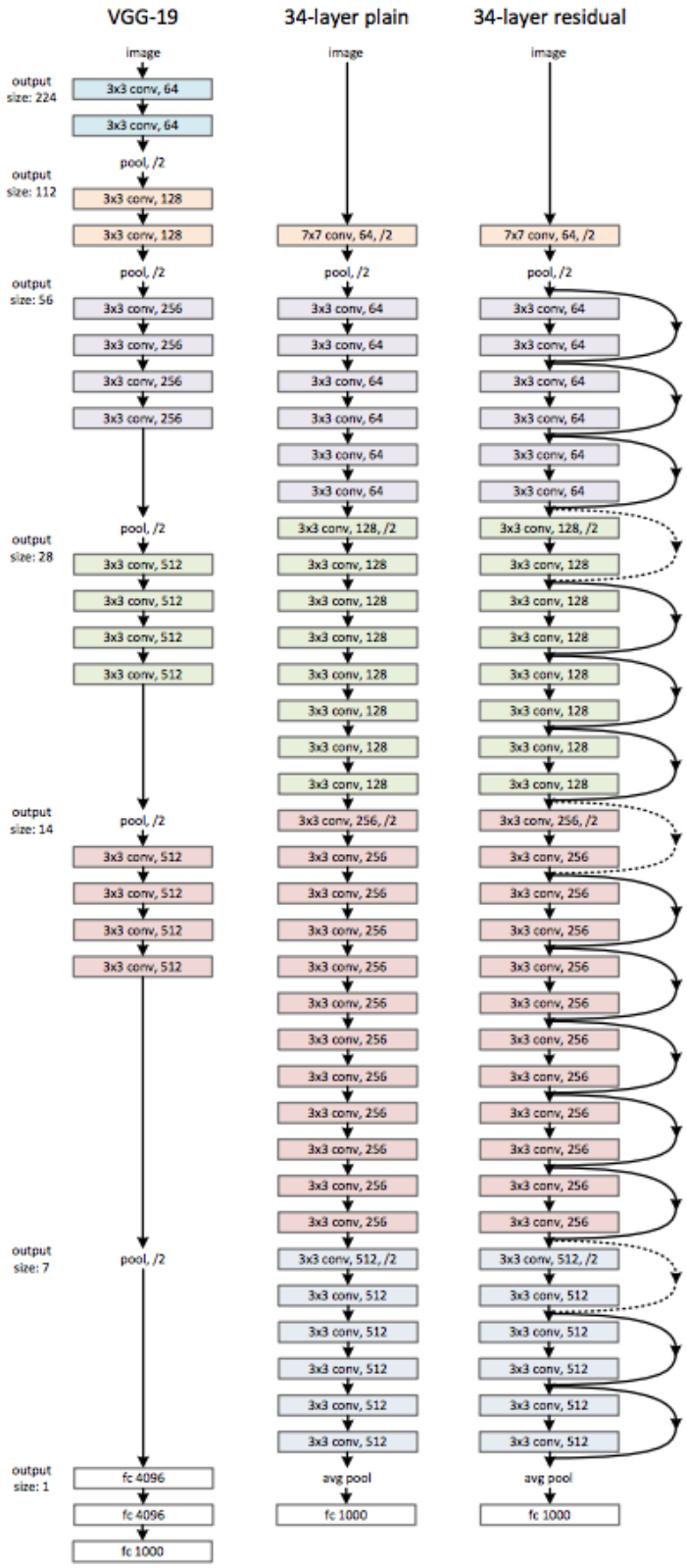
$$\begin{aligned}\frac{\partial C}{\partial x} &= \frac{\partial C}{\partial H}\frac{\partial H}{\partial x} \\ &= \frac{\partial C}{\partial H}(\frac{\partial F}{\partial x} + 1) \\ &= \frac{\partial C}{\partial H}\frac{\partial F}{\partial x} + \frac{\partial C}{\partial H}\end{aligned}$$

(A minor point: we can use a learned matrix in case x and $F(x)$ are of a different dimensionality to make them compatible for addition.)

With that addition, the gradient is less likely to go to zero and we simply propagate the complete gradient backwards. These residual connections act as a "gradient highway" since the gradient distributes evenly at sums in a computation graph. This allows us to preserve the gradient as we go backwards.

Additionally, we can have **bottleneck residual blocks**, as shown on the right-side of the figure. We use those 1x1 convolutions again to reduce dimensionality before and after the middle convolutional layer.

It turns out that these residual blocks are so powerful that we can stack many of these to produce networks that are over 5 times deeper than before! The deepest variant of ResNet was ResNet 151. That's *151 layers deep!* Before that, GoogLeNet was a mere 22 layers deep!

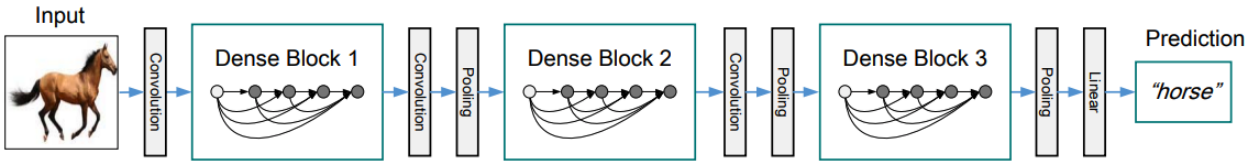


Source (*Deep Residual Learning for Image Recognition* by He et al.)

ResNets revolutionized deep architectures and now researchers are starting to use skip connections to make their architectures deeper. ResNets *swept* the ILSVRC in 2015!

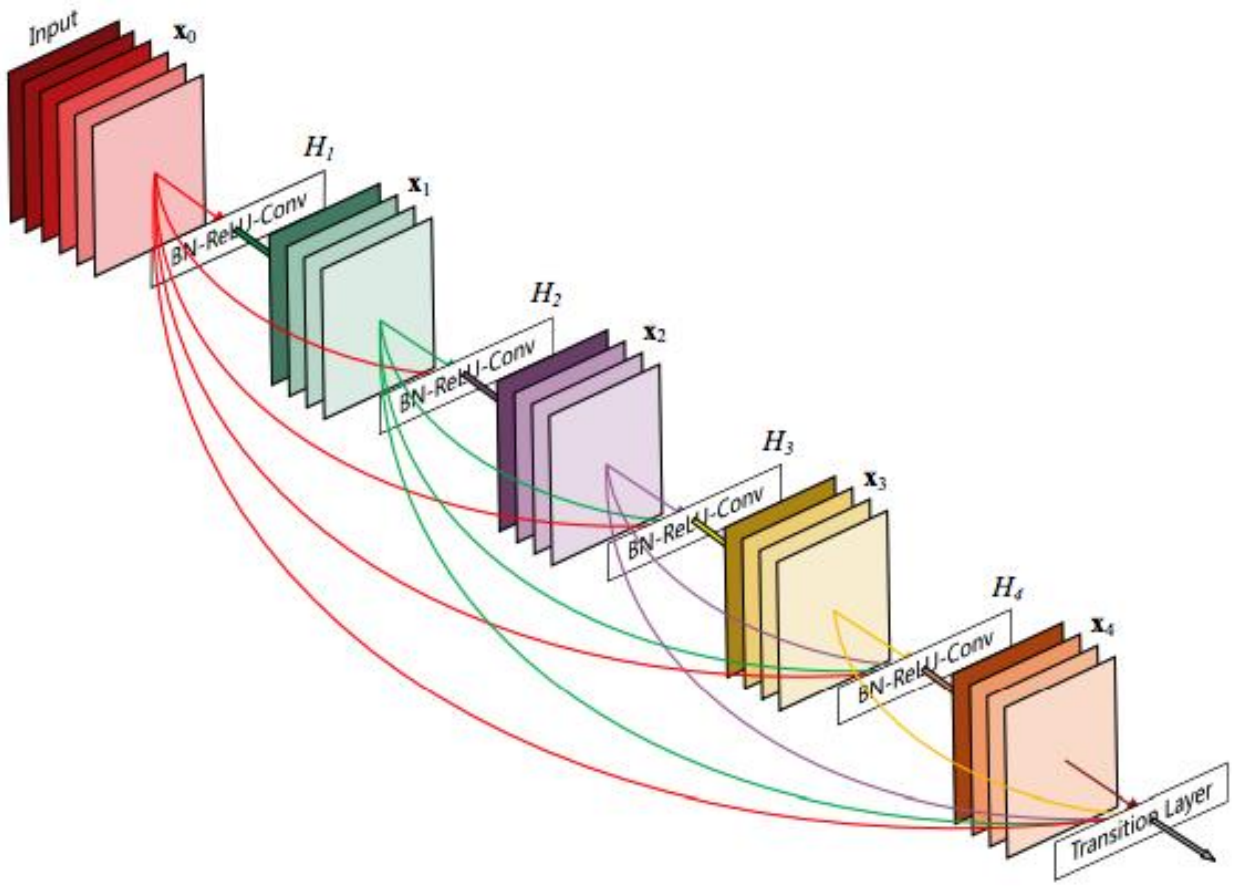
DenseNet

The most recent new architecture is from Facebook AI Research (FAIR) and won best paper at the most prestigious computer vision conference: Computer Vision and Pattern Recognition (CVPR) in 2017. Their architecture was titled **DenseNet**. Like GoogLeNet and ResNet before it, DenseNet introduced a new block called a **Dense Block** and stacked these blocks on top of each other, with some layers in between, to build a deep network.



Source (*Densely Connected Convolutional Networks* by Huang et al.)

These dense blocks take the concept of residual networks a step further and connect every layer to every other layer! In other words, for a dense block, we consider all dense block before it as input and we produce an output that we feed into all subsequent dense blocks. To make the layers compatible with each other, we apply convolutions and batch normalizations. The benefit of doing this is that we encourage feature reuse, resolve the vanishing gradient problem, and (counter-intuitively!) have fewer parameters overall!



Source (*Densely Connected Convolutional Networks* by Huang et al.)

As mentioned before, DenseNets won the Best Paper award at CVPR 2017 and achieved state-of-the-art performance on another image classification challenge: CIFAR-1000. Additionally, their work showed that DenseNets actually have fewer than half the number of parameters than ResNets and are comparable in performance on ImageNet.

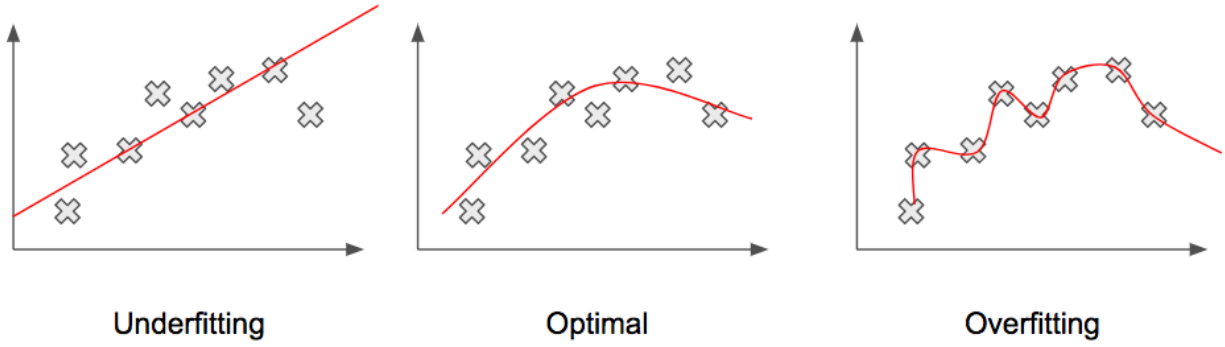
To summarize, we discussed several different architectures that are being used today in convolutional neural networks for image classification. We discussed a few of the early deep learning architectures, such as AlexNet and VGG Net. Then, we discussed Google's architecture that used the inception modules called GoogLeNet that won the ILSVRC in 2014. After that, we discussed the deepest network to date: ResNets. Using these residual connections, we can mitigate the vanishing gradient problem and stack these together to get networks that are over 100 layers deep! Finally, we discussed a very recent paper called DenseNet that exploited densely connections between their Dense Blocks.

New deep convolutional neural network architectures are being created each day and tested for efficiency against these. Keep up-to-date on these cutting-edge new architectures!

A Guide to Improving Deep Learning's Performance

Although deep learning has great potential to produce fantastic results, we can't simply leave everything to the learning algorithm! In other words, we can't treat the model as some black-box, closed entity that can read our minds and perform the best! We have to be involved in the training and design process to make sure our model is learning efficiently. We're going to look at several different techniques we can apply to every deep learning model that will help improve our model's accuracy.

Overfitting and Underfitting



To discuss overfitting and underfitting, let's consider the challenge of curve-fitting: given a set of points, find the curve of best fit. We might think that the curve that goes through all of the points is the *best* curve, but, actually, that's not quite right! If we gave new data to that curve, it wouldn't do well! This problem is called **overfitting**. Why we're discussing it is because it is *very common* in deep architectures. Overfitting happens when our model tries so hard to correctly classify each and every example, that it ends up modeling of all of these tiny noises and intricacies for each input. Then, when we give it new data it hasn't seen before, the model doesn't know what to do! We say it **generalizes** poorly! We want our model to generalize to new, never-before-seen data. If it overfits, then we'll get poor accuracy on new data.

Overfitting is also related to the size of the model and is, therefore, a *huge* problem in deep learning where we have millions of parameters! The more parameters we have, the better we can fit the data. Specifically for curve-fitting, we can perfectly fit a curve to N number of points using an N-1 degree polynomial.

There are several ways to detect overfitting. We can plot the error/loss of the training data and the same for the validation set. If we see that our training loss is very small, but our validation set loss is still high, this is an indication of overfitting. Our model is doing really well on the training set but is failing to generalize. Another, similar, indication is the training set and testing set accuracy. If our model has a very high training set accuracy and very low test set

accuracy, this is an indication of overfitting for the same reason: our model isn't generalizing! To combat overfitting, we use **regularization**. We'll discuss a few techniques later.

The reciprocal case to overfitting is **underfitting**. This is less of a problem in deep learning but does help with model selection. In the case of **underfitting**, our model is generalizing so much that it's actually missing the underlying relationship of the data. In the figure above, the line is linear when the data are clearly non-linear. This type of generalization also leads to poor accuracy!

To detect underfitting, we can look at the training and validation loss. If both are high, then we know that our model is underfitting. To prevent underfitting, we can simply use a larger model! If we have a 2-layer network, we can try increasing the number of hidden neurons or try adding more hidden layers. Both of these will help increase the number of parameters of our model and prevent underfitting.

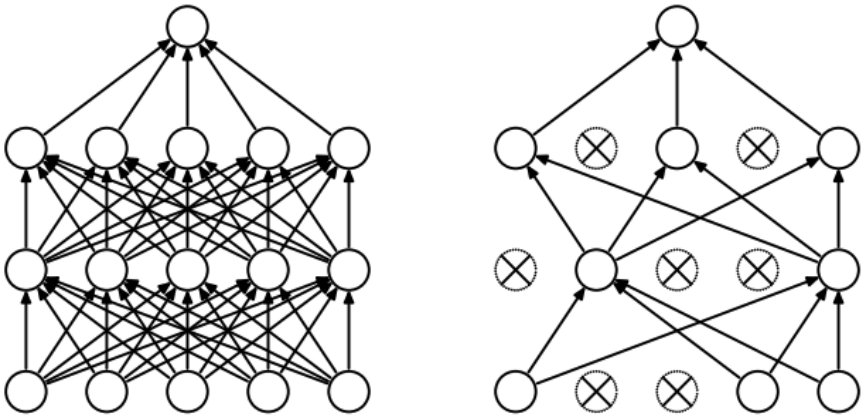
This problem of overfitting and underfitting is called the **bias-variance dilemma** or **tradeoff** in learning theory. We're referring to the bias and variance *of the model*, not the actual bias or variance of the parameters! The bias and variance represent how much we pay attention to the model data itself. If we have a **high-variance** model, this means we're paying too much attention to the intricacies of the data. A **high-bias** model means we're ignoring the data entirely. The goal is to build a model with low bias and low variance.

We can explain overfitting and underfitting in terms of bias and variance. If we have an overfit model, then we say this is a **high-variance model**. This is because the overfit model is learning the tiny intricacies and noise of the data. An underfit model is a **high-bias model** because we're completely missing and ignoring the underlying structure of the data.

To summarize, we can look at the training and testing loss to determine if our system is underfitting or overfitting. If both losses are high, then our model is underfitting, and we need to increase the number of parameters. If our training loss is low and our testing loss is high, then our model is overfitting and not generalizing well. In this case, we can use **regularization** to help prevent overfitting. Let's discuss a few techniques!

Dropout

Dropout is a technique that was designed to counteract overfitting. In fact, the paper that introduced it is titled *Dropout: A Simple Way to Prevent Neural Networks from Overfitting* by Srivastava et al. In the paper, they present a radical new way to prevent overfitting and improve generalization.



(a) Standard Neural Net (b) After applying dropout.

Figure 1: Dropout Neural Net Model. **Left**: A standard neural net with 2 hidden layers. **Right**: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

(*Dropout: A Simple Way to Prevent Neural Networks from Overfitting* by Srivastava et al.)

The idea behind dropout is to randomly zero out some of the weights in a given layer. When we do this, we effectively "kill" that neuron. Then, we continue with the regular training process for each example in a mini-batch. The figure above is taken from the paper and pictorially explains this nullification.

Mathematically, each layer that has dropout enabled, which won't be all layers, will have dropout neurons that each have some equal probability of being dropped out. For example, in the above 2-layer network, each neuron in the second hidden layer has a probability of being dropped out. For each example in each mini-batch, we drop out some neurons, pass the example forward, *and backpropagate with the neurons still dropped out!* This is the critical point: the neurons that were dropped out in the forward pass are *also dropped out in the backward pass*. Remember that neural networks learn from backpropagation. If we didn't also drop out the same neurons during the backward pass, then dropout wouldn't do anything! During testing, however, we enable all of the neurons.

This was a bit of a radical technique when it was introduced. Many researchers asked, "how could something like this *help* the network learn?" There is an intuition behind why this works: by *thinning* the network, preventing it from using all of the neurons, we force it to learn alternative or redundant representations because the network can never know which neurons will be unavailable for it to use.

As great as this technique sounds, let's not be dropout-happy and apply it to each and every layer! There are a few instances where we don't use dropout. For example, we never

apply dropout in the output layer. This is because the output neurons produce a probability distribution over all of the classes. It doesn't make sense to drop out any neurons in this layer because then we're randomly ignoring some classes!

Dropout has implementations in almost all major libraries, such as Tensorflow, Keras, and Torch. For example, the following line of code in Keras will add Dropout to the previous layer with a dropout probability of 0.5.

```
model.add(Dropout(0.5))
```

In most cases, a dropout probability of 0.5 is used. This regularization technique of dropout is *very widely* used in almost all deep learning architectures!

There is one minor point to discuss when implementing dropout: output scaling. During training, we have to scale the outputs of the neurons. This is because we need to ensure that the training and testing phases' activations are identically scaled, because, during testing, all neurons receive all inputs while, during training, a fraction of neurons see the inputs. However, since testing-time performance is very important, we often implement dropout as **inverted dropout** where we scale during *training* instead of testing.

Weight Regularization

There are two other types of regularization that are a function of the parameters or weights themselves. We call these ℓ_1 and ℓ_2 regularization. These correspond intuitively to weight sparsity and weight decay. Both of these regularizers are attached to the cost function with a parameter λ that controls how much emphasis we should put on the regularization.
$C = E + \lambda R(W)$

Here, C is a cost function where E computes the error using something like squared error, cross entropy, etc. However, we introduce another function $R(W)$ that is the regularizer and the λ parameter. Since our new cost function is a sum of two quantities with a preference parameter λ, our optimizer is going to perform a balancing act: minimize the error while also obeying the regularizer. In other words, the regularizer expresses a *preference* to do something. In the case of regularization, we're going to prefer the optimizer to do something with respect to the weights. Let's discuss the types of regularization!

We'll first discuss ℓ_1 **regularization**, shown below. This is also called weight sparsity because of what it tries to do: make as many weight values 0 as possible. The idea is to add a quantity to the cost function penalizes weights that have nonzero values. Notice that $R(W) = 0$ when all of the weights are zero. However, the network wouldn't learn anything in this case! This prevents us from ever reaching the case where $R(W) = 0$. In other words, the ℓ_1 regularizer

prefers many zero weights. This is called weight sparsity. An interesting product of ℓ_1 regularization is that it acts as a *feature selector*. In other words, due to sparsity and the preference to keep zero weights, the weights that are nonzero are very important because they would have been set to zero by the optimizer otherwise!

$$R(W) = \sum_k |w_k|$$

Now let's move on to ℓ_2 **regularization**, shown below. This is sometimes called weight decay. Although they are technically different concepts, the former being a quantity added to the cost function and the latter being a factor applied to the update rule, they have the same purpose: *penalize large weights!* This regularizer penalizes large weight values by adding the squared size of all of the weights to the cost function. This forces our optimization algorithm to try to minimize the sum of the squared size of all of the weights, i.e., bring them closer to zero. In other words, we want to avoid the scenario where a few weights have really large values and the rest of the weights are zero. We want the scenario where the weight values are fairly spread out among the weights.

$$R(W) = \sum_k w_k^2$$

One more point regarding weight decay and ℓ_2 regularization. Although they are conceptually different, mathematically, they are equivalent. By introducing ℓ_2 regularization, we're decaying the weights linearly by a constant factor during gradient descent, which is exactly weight decay.

Like dropout, all major libraries support regularization. In Keras, we can add regularization to weights like this.

```
model.add(Dense(1024, kernel_regularizer=regularizers.l2(0.01))
```

In this case, we're using ℓ_2 Regularization with a strength of 0.01. In practice, the most common kind of regularization used is ℓ_2 Regularization and Dropout. Using these techniques, we can prevent our model from overfitting and help it generalize better!

We've covered some important aspects of applied deep learning. We discussed this problem of underfitting and overfitting and the bias-variance dilemma/tradeoff. Overfitting is more common in deep learning because we often deal with models with millions of parameters. We discussed a technique called Dropout that can help us prevent overfitting. The idea is to drop out neurons for each dropout layer. This forces our network to learn redundancies when training. During testing, we give all neurons all inputs again. We also looked at two other types of regularization that produce interesting weight characteristics: ℓ_1 and ℓ_2 regularization. The former produces sparse weights and acts a feature selector: we have many zero weights, and the nonzero weights are the most important features. With the latter flavor of regularization, we

prefer diffuse weights to a few weights with large values. This is similar to the technique of weight decay and reduces to the same thing mathematically.

The concept of regularization is widely used in deep learning and critical to preventing model overfitting!

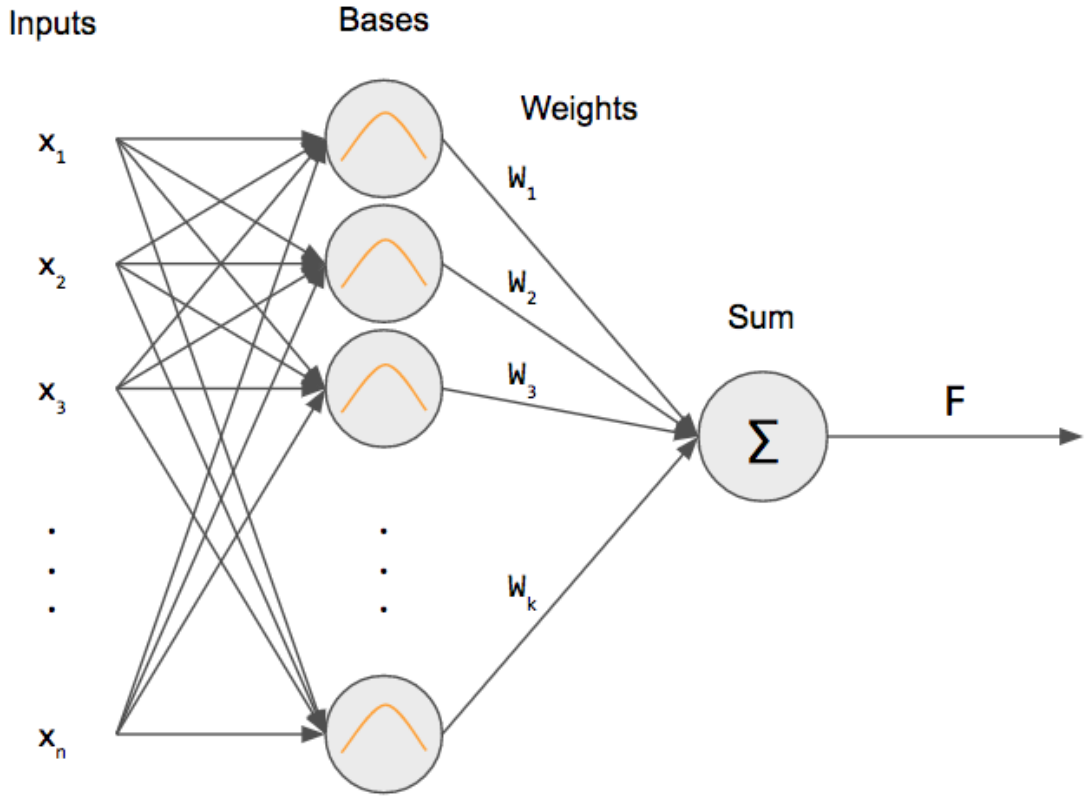
Using Neural Networks for Regression: Radial Basis Function Networks

Neural Networks are very powerful models for classification tasks. But what about regression? Suppose we had a set of data points and wanted to project that trend into the future to make predictions. Regression has many applications in finance, physics, biology, and many other fields.

Radial Basis Function Networks (RBF nets) are used for exactly this scenario: regression or function approximation. We have some data that represents an underlying trend or function and want to model it. RBF nets can learn to approximate the underlying trend using many Gaussians/bell curves.

Before we begin, please familiarize yourself with neural networks, backpropagation, and k-means clustering.

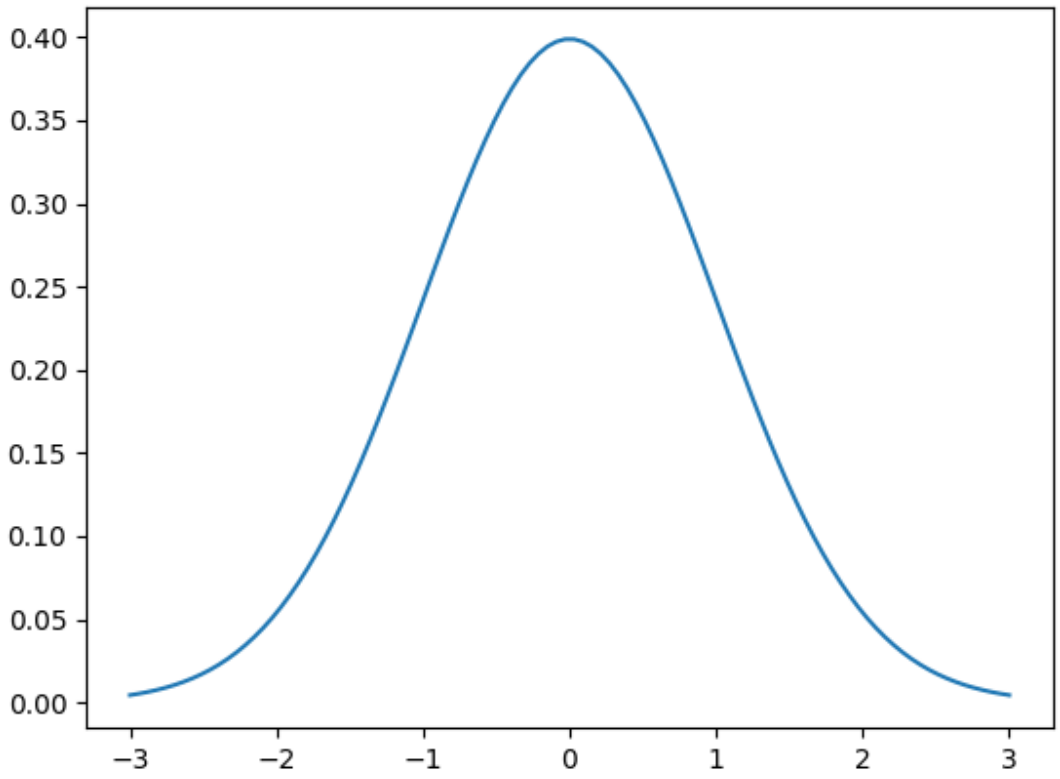
An RBF net is similar to a 2-layer network. We have an input that is fully connected to a hidden layer. Then, we take the output of the hidden layer perform a weighted sum to get our output.



But what is that inside the hidden layer neurons? That is a Gaussian RBF! This differentiates an RBF net from a regular neural network: we're using an RBF as our "activation" function (more specifically, a Gaussian RBF).

Gaussian Distribution

The first question you may have is "what is a Gaussian?" It's the most famous and important of all statistical distributions. A picture is worth a thousand words so here's an example of a Gaussian centered at 0 with a standard deviation of 1.



This is the **Gaussian** or **normal distribution**! It is also called a **bell curve** sometimes. The function that describes the normal distribution is the following

$$\mathcal{N}(x; \mu, \sigma^2) = displaystyle \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

That looks like a really messy equation! And it is, so we'll use $\mathcal{N}(x; \mu, \sigma^2)$ to represent that equation. If we look at it, we notice there are one input and two parameters. First, let's discuss the parameters and how they change the Gaussian. Then we can discuss what the input means.

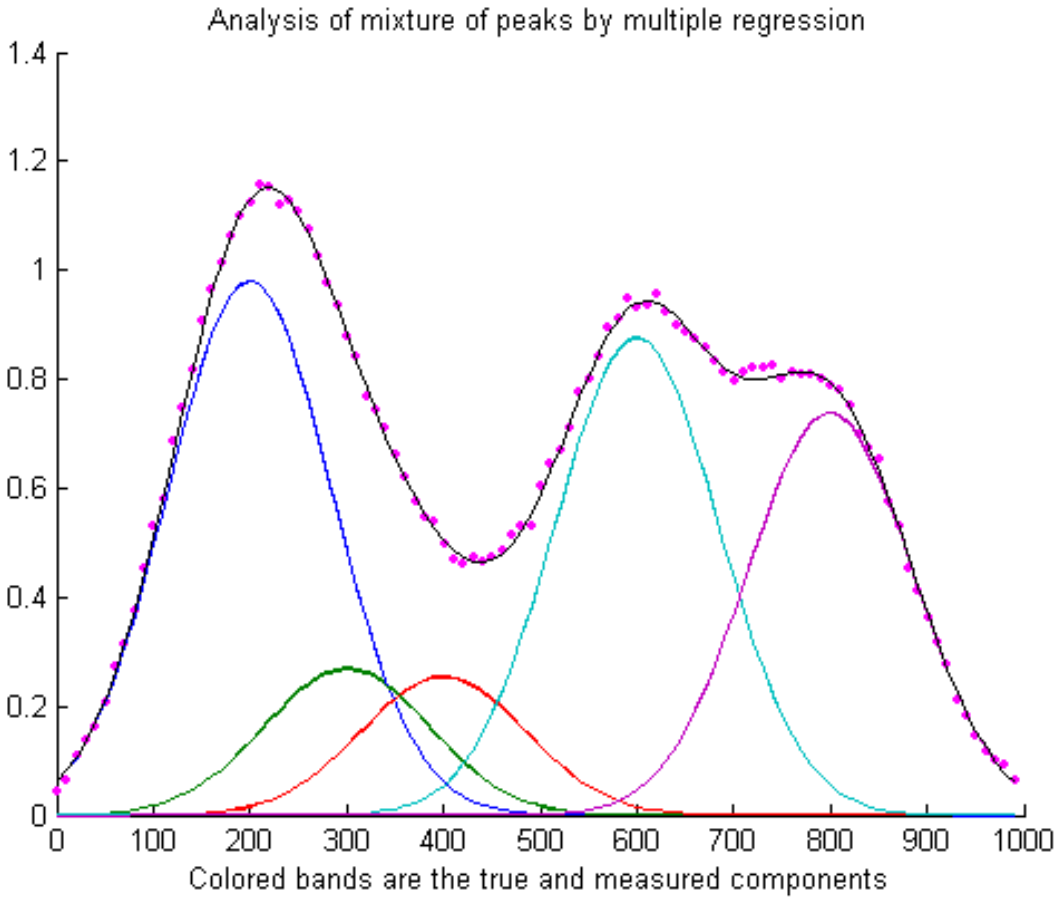
The two parameters are called the **mean** μ and **standard deviation** σ. In some cases, the standard deviation is replaced with the **variance** σ^2, which is just the square of the standard deviation. The mean of the Gaussian simply shifts the center of the Gaussian, i.e. the "bump" or top of the bell. In the image above, $\mu = 0$, so the largest value is at $x = 0$.

The standard deviation is a measure of the *spread* of the Gaussian. It affects the "wideness" of the bell. Using a larger standard deviation means that the data are more spread out, rather than closer to the mean.

Technically, the above function is called the **probability density function (pdf)** and it tells us the probability of observing an input x, given that specific normal distribution. But we're only interested in the bell-curve properties of the Gaussian, not the fact that it represents a probability distribution.

Gaussians in RBF nets

Why do we care about Gaussians? We can use a *linear combination* of Gaussians to approximate any function!



Source: https://terpconnect.umd.edu/~toh/spectrum/CurveFittingB.html

In the figure above, the Gaussians have different colors and are weighted differently. When we take the sum, we get a continuous function! To do this, we need to know where to place the Gaussian centers c_j and their standard deviations σ_j.

We can use k-means clustering on our input data to figure out where to place the Gaussians. The reasoning behind this is that we want our Gaussians to "span" the largest clusters of data since they have that bell-curve shape.

The next step is figuring out what the standard deviations should be. There are two approaches we can take: set the standard deviation to be that of the points assigned to a particular cluster c_j or we can use a single standard deviation for all clusters $\sigma_j = \sigma \forall j$ where where $\sigma = \frac{d_{\max}}{\sqrt{2k}}$ where $d_{\max}$ is the maximum distance between any two cluster centers. and k is the number of cluster centers.

But wait, how many Gaussians do we use? Well that's a hyperparameter called the number of **bases** or **kernels** k.

Backpropagation for RBF nets

K-means clustering is used to determine the centers c_j for each of the radial basis functions φ_j. Given an input x, an RBF network produces a weighted sum output.

$$F(x) = \sum_{j=1}^{k} w_j \varphi_j(x, c_j) + b$$

where w_j are the weights, b is the bias, k is the number of bases/clusters/centers, and $\varphi_j(\cdot)$ is the Gaussian RBF:

$$\varphi_j(x, c_j) = \exp\left(\frac{-||x - c_j||^2}{2\sigma_j^2}\right)$$

There are other kinds of RBFs, but we'll stick with our Gaussian RBF. (Notice that we don't have the constant up front, so our Gaussian is not normalized, but that's ok since we're not using it as a probability distribution!)

Using these definitions, we can derive the update rules for w_j and b for gradient descent. We use the quadratic cost function to minimize.

$$C = \sum_{i=1}^{N} (y^{(i)} - F(x^{(i)}))^2$$

We can derive the update rule for w_j by computing the partial derivative of the cost function with respect to all of the w_j.

$$\begin{aligned}
\frac{\partial C}{\partial w_j} &= \frac{\partial C}{\partial F}\frac{\partial F}{\partial w_j} \\
&= \frac{\partial}{\partial F}[\sum_{i=1}^{N}(y^{(i)} - F(x^{(i)}))^2] \ \cdot \ \frac{\partial}{\partial w_j}[\sum_{j=0}^{K} w_j \varphi_j(x, c_j) + b] \\
&= -(y^{(i)} - F(x^{(i)})) \ \cdot \ \varphi_j(x, c_j) \\
w_j &\leftarrow w_j + \eta \ (y^{(i)} - F(x^{(i)}))\ \varphi_j(x, c_j)
\end{aligned}$$

Similarly, we can derive the update rules for b by computing the partial derivative of the cost function with respect to b.

$$\begin{aligned}
\frac{\partial C}{\partial b} &= \frac{\partial C}{\partial F}\frac{\partial F}{\partial b} \\
&= \frac{\partial}{\partial F}[\sum_{i=1}^{N}(y^{(i)} - F(x^{(i)}))^2] \ \cdot \ \frac{\partial}{\partial b}[\sum_{j=0}^{K} w_j \varphi_j(x, c_j) + b] \\
&= -(y^{(i)} - F(x^{(i)})) \cdot 1 \\
b &\leftarrow b + \eta \ (y^{(i)} - F(x^{(i)}))
\end{aligned}$$

Now we have our backpropagation rules!

RBF Net Code

Now that we have a better understanding of how we can use neural networks for function approximation, let's write some code!

First, we have to define our "training" data and RBF. We're going to code up our Gaussian RBF.

```
def rbf(x, c, s):
```

```
    return np.exp(-1 / (2 * s**2) * (x-c)**2)
```

Now we'll need to use the k-means clustering algorithm to determine the cluster centers. I've already coded up a function for you that gives us the cluster centers and the standard deviations of the clusters.

```
def kmeans(X, k):
    """Performs k-means clustering for 1D input

    Arguments:
        X {ndarray} -- A Mx1 array of inputs
        k {int} -- Number of clusters

    Returns:
        ndarray -- A kx1 array of final cluster centers
    """

    # randomly select initial clusters from input data
    clusters = np.random.choice(np.squeeze(X), size=k)
    prevClusters = clusters.copy()
    stds = np.zeros(k)
    converged = False

    while not converged:
        """
        compute distances for each cluster center to each point
        where (distances[i, j] represents the distance between the ith
point and jth cluster)
        """
        distances = np.squeeze(np.abs(X[:, np.newaxis] -
clusters[np.newaxis, :]))

        # find the cluster that's closest to each point
        closestCluster = np.argmin(distances, axis=1)

        # update clusters by taking the mean of all of the points assigned
to that cluster
        for i in range(k):
            pointsForCluster = X[closestCluster == i]
```

```
            if len(pointsForCluster) > 0:
                clusters[i] = np.mean(pointsForCluster, axis=0)

        # converge if clusters haven't moved
        converged = np.linalg.norm(clusters - prevClusters) < 1e-6
        prevClusters = clusters.copy()

    distances = np.squeeze(np.abs(X[:, np.newaxis] - clusters[np.newaxis,
:]))
    closestCluster = np.argmin(distances, axis=1)

    clustersWithNoPoints = []
    for i in range(k):
        pointsForCluster = X[closestCluster == i]
        if len(pointsForCluster) < 2:
            # keep track of clusters with no points or 1 point
            clustersWithNoPoints.append(i)
            continue
        else:
            stds[i] = np.std(X[closestCluster == i])

    # if there are clusters with 0 or 1 points, take the mean std of the
other clusters
    if len(clustersWithNoPoints) > 0:
        pointsToAverage = []
        for i in range(k):
            if i not in clustersWithNoPoints:
                pointsToAverage.append(X[closestCluster == i])
        pointsToAverage = np.concatenate(pointsToAverage).ravel()
        stds[clustersWithNoPoints] = np.mean(np.std(pointsToAverage))

    return clusters, stds
```

This code just implements the k-means clustering algorithm and computes the standard deviations. If there is a cluster with none or one assigned points to it, we simply average the standard deviation of the other clusters. (We can't compute standard deviation with no data points, and the standard deviation of a single data point is 0).

We're not going to spend too much time on k-means clustering. Visit the link at the top for more information.

Now we can get to the real heart of the RBF net by creating a class.

```
class RBFNet(object):
    """Implementation of a Radial Basis Function Network"""
    def __init__(self, k=2, lr=0.01, epochs=100, rbf=rbf, inferStds=True):
        self.k = k
        self.lr = lr
        self.epochs = epochs
        self.rbf = rbf
        self.inferStds = inferStds

        self.w = np.random.randn(k)
        self.b = np.random.randn(1)
```

We have options for the number of bases, learning rate, number of epochs, which RBF to use, and if we want to use the standard deviations from k-means. We also initialize the weights and bias. Remember that an RBF net is a modified 2-layer network, so there's only only one weight vector and a single bias at the output node, since we're approximating a 1D function (specifically, one output). If we had a function with multiple outputs (a function with a vector-valued output), we'd use multiple output neurons and our weights would be a matrix and our bias a vector.

Then, we have to write our fit function to compute our weights and biases. In the first few lines, we either use the standard deviations from the modified k-means algorithm, or we force all bases to use the same standard deviation computed from the formula. The rest is similar to backpropagation where we propagate our input going forward and update our weights going backward.

```
def fit(self, X, y):
    if self.inferStds:
        # compute stds from data
        self.centers, self.stds = kmeans(X, self.k)
    else:
        # use a fixed std
        self.centers, _ = kmeans(X, self.k)
        dMax = max([np.abs(c1 - c2) for c1 in self.centers for c2 in
self.centers])
        self.stds = np.repeat(dMax / np.sqrt(2*self.k), self.k)
```

```
    # training
    for epoch in range(self.epochs):
        for i in range(X.shape[0]):
            # forward pass
            a = np.array([self.rbf(X[i], c, s) for c, s, in
zip(self.centers, self.stds)])
            F = a.T.dot(self.w) + self.b

            loss = (y[i] - F).flatten() ** 2
            print('Loss: {0:.2f}'.format(loss[0]))

            # backward pass
            error = -(y[i] - F).flatten()

            # online update
            self.w = self.w - self.lr * a * error
            self.b = self.b - self.lr * error
```

For verbosity, we're printing the loss at each step. Notice we're also performing an online update, meaning we update our weights and biases each input. Alternatively, we could have done a batch update, where we update our parameters after seeing all training data, or minibatch update, where we update our parameters after seeing a subset of the training data. Making a prediction is as simple as propagating our input forward.

```
def predict(self, X):
    y_pred = []
    for i in range(X.shape[0]):
        a = np.array([self.rbf(X[i], c, s) for c, s, in zip(self.centers,
self.stds)])
        F = a.T.dot(self.w) + self.b
        y_pred.append(F)
    return np.array(y_pred)
```

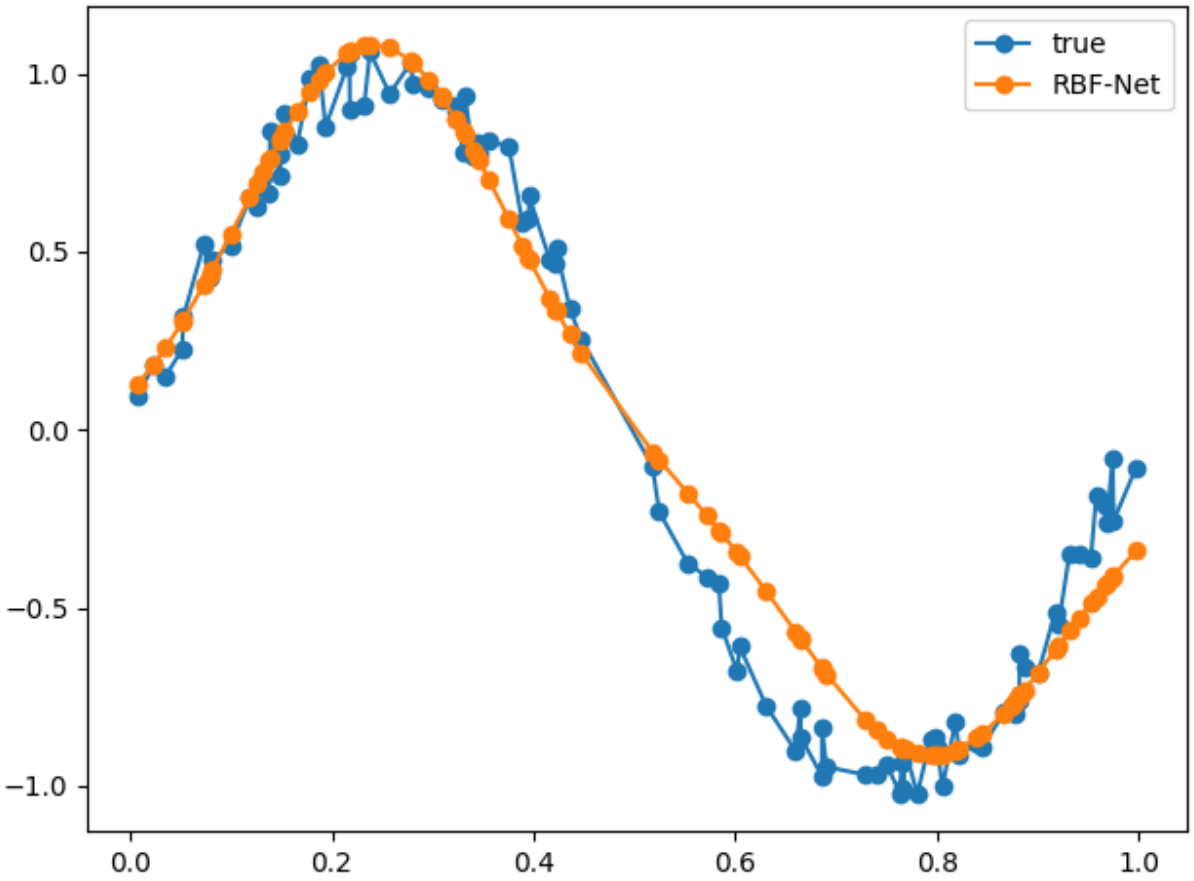
Notice that we're allowing for a matrix inputs, where each row is an example.

Finally, we can write code to use our new class. For our training data, we'll be generating 100 samples from the sine function. Then, we'll add some uniform noise to our data.

```
# sample inputs and add noise
```

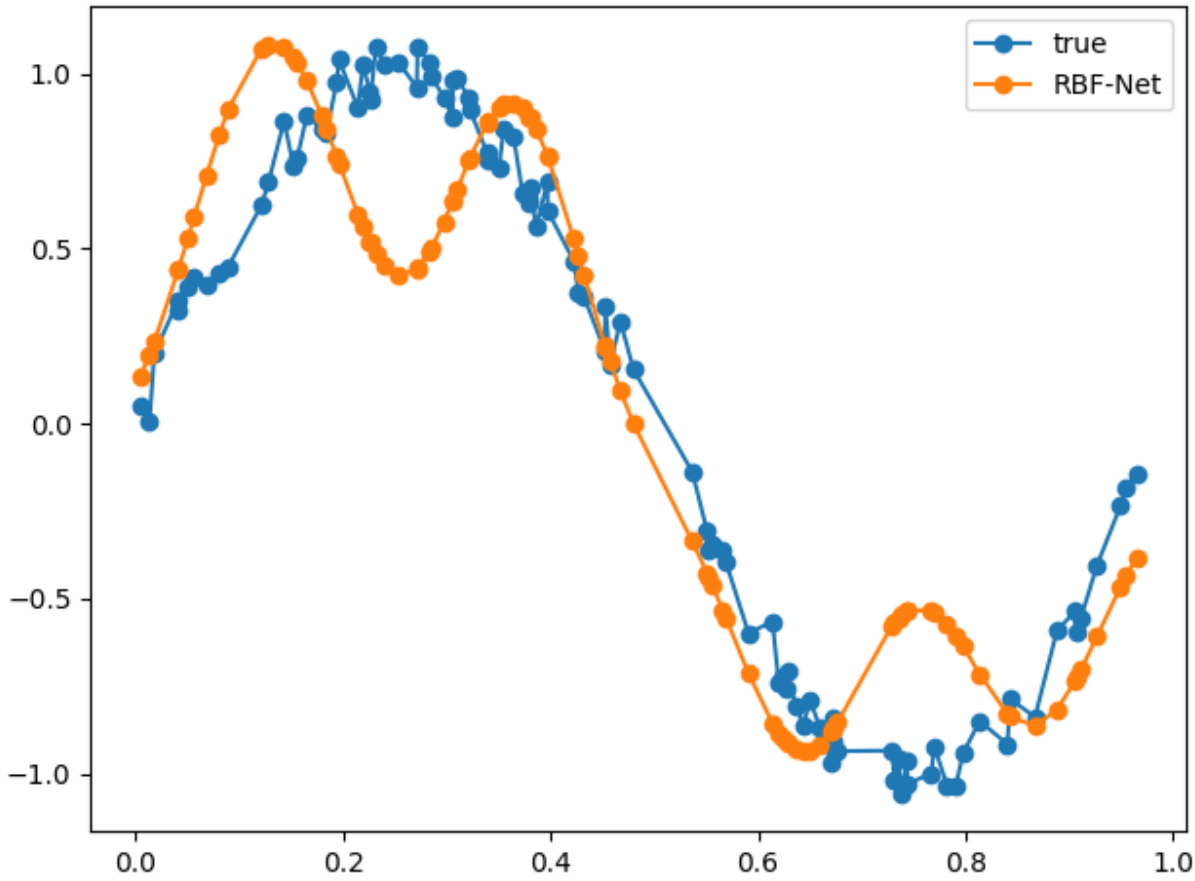
```
NUM_SAMPLES = 100
X = np.random.uniform(0., 1., NUM_SAMPLES)
X = np.sort(X, axis=0)
noise = np.random.uniform(-0.1, 0.1, NUM_SAMPLES)
y = np.sin(2 * np.pi * X)  + noise

rbfnet = RBFNet(lr=1e-2, k=2)
rbfnet.fit(X, y)

y_pred = rbfnet.predict(X)

plt.plot(X, y, '-o', label='true')
plt.plot(X, y_pred, '-o', label='RBF-Net')
plt.legend()

plt.tight_layout()
plt.show()
```

We can plot our approximated function against our real function to see how well our RBF net performed.



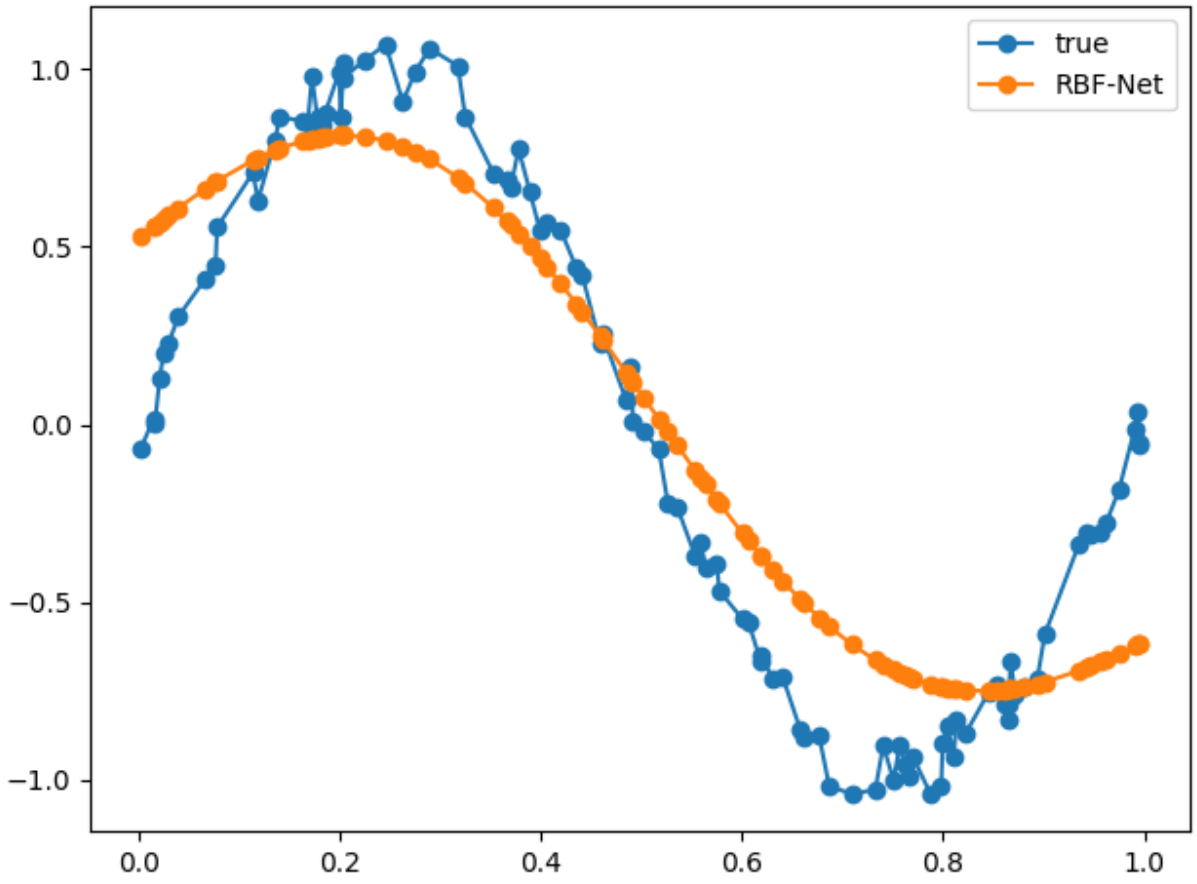
From our results, our RBF net performed pretty well! If we wanted to evaluate our RBF net more rigorously, we could sample more points from the same function, pass it through our RBF net and use the summed Euclidean distance as a metric.

We can try messing around with some key parameters, like the number of bases. What if we increase the number of bases to 4?



Our results aren't too great! This is because our original function is shaped the way that it is, i.e., two bumps. If we had a more complicated function, then we could use a larger number of bases. If we used a large number of bases, then we'll start overfitting!

Another parameter we can change is the standard deviation. How about we use a single standard deviation for all of our bases instead of each one getting its own?



Our plot is much smoother! This is because the Gaussians that make up our reconstruction all have the same standard deviation.

There are other parameters we can change like the learning rate; we could use a more advanced optimization algorithm; we could try layering Gaussians; etc.

To summarize, RBF nets are a special type of neural network used for regression. They are similar to 2-layer networks, but we replace the activation function with a radial basis function, specifically a Gaussian radial basis function. We take each input vector and feed it into each basis. Then, we do a simple weighted sum to get our approximated function value at the end. We train these using backpropagation like any neural network! Finally, we implemented RBF nets in a class and used it to approximate a simple function.
RBF nets are a great example of neural models being used for regression!

All about autoencoders

Data compression is a big topic that's used in computer vision, computer networks, computer architecture, and many other fields. The point of data compression is to convert our input into a *smaller* representation that we recreate, to a degree of quality. This smaller representation is what would be passed around, and, when anyone needed the original, they would reconstruct it from the smaller representation.

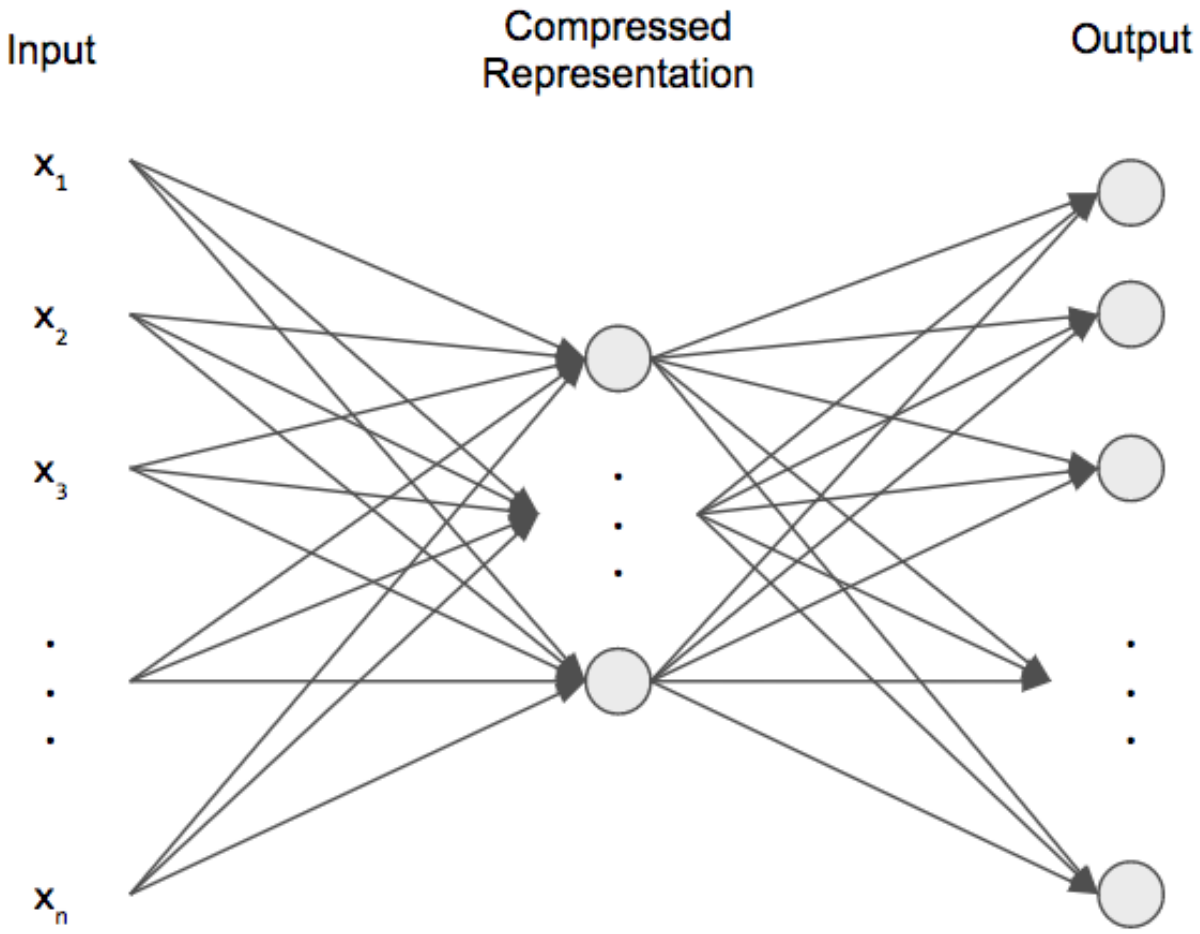
Consider a ZIP file. When we create a ZIP file, we compress our files so that they take up fewer bytes. Then we pass around that ZIP file. If we wanted to access the contents, we can uncompress the ZIP file, and reconstruct the contents from the ZIP file.
In another example, consider a JPEG image file. This is an example of a lossy format: when we compress a JPEG, we lose information about the original. If we uncompress it, then our reconstruction isn't perfect. However, for JPEG, we can compress it down to a *tenth* of the original data without any noticeable loss in image quality!

Many techniques in the past have been hard-coded or use clever algorithms. **Autoencoders** are *unsupervised neural networks* that use machine learning to do this compression for us. There are many different kinds of autoencoders that we're going to look at: vanilla autoencoders, deep autoencoders, deep autoencoders for vision. Finally, we'll apply autoencoders for removing noise from images.

Vanilla Autoencoder

We'll first discuss the simplest of autoencoders: the standard, run-of-the-mill autoencoder. Essentially, an autoencoder is a 2-layer neural network that satisfies the following conditions.

1. The hidden layer is *smaller* than the size of the input and output layer.
2. The input layer and output layer are the same size.



The hidden layer is **compressed representation**, and we learn two sets of weights (and biases) that *encode* our input data into the compressed representation and *decode* our compressed representation back into input space.

Notice that there are no labels! Our input and output are the same! But then what is our loss function? We have a simple Euclidean distance loss: $||\mathbf{x} - \hat{\mathbf{x}}||^2$ called the **reconstruction error**, where the input is $\mathbf{x}$ and the reconstruction is $\hat{\mathbf{x}}$. We want to minimize this error. In other words, this error represents how close our reconstruction was to the true input data. We won't expect a perfect reconstruction since the number of hidden neurons is less than the number of input neurons, but we want the parameters to give us the *best* possible reconstruction.

Mathematically, our above autoencoder can be thought of as two separate things: an encoder and decoder.

$$\mathbf{z} = \sigma(\mathbf{W}^{(e)}\mathbf{x} + \mathbf{b}^{(e)})$$
$$\hat{\mathbf{x}} = \sigma(\mathbf{W}^{(d)}\mathbf{z} + \mathbf{b}^{(d)})$$

where the superscripts correspond to the encoder and decoder and the input is $\mathbf{x}$. Hence, our loss function will be the squared Euclidean error.
$\mathcal{L}(\mathbf{x}, \hat{\mathbf{x}}) = ||\mathbf{x} - \hat{\mathbf{x}}||_2^2$

When we train our autoencoder, we're trying to minimize $\mathcal{L}$. We won't see the backpropagation derivation of the update rules, but they're identical to a standard neural network.

The big catch with autoencoders is that they only work for the data we train them on! If we train our autoencoder on images of cats, then it won't work too well for images of dogs! This is all there is to autoencoders! They're simple neural networks but also very powerful! Let's code up an autoencoder.

First, we'll need to load our MNIST handwritten digits dataset. Notice that we're not loading any of the labels because autoencoders are *unsupervised*. Additionally, we rescale our images from 0 - 255 to 0 - 1 and flatten them out.

```
from keras.layers import Input, Dense
from keras.models import Model
from keras.datasets import mnist

import numpy as np
import matplotlib.pyplot as plt

(X_train, _), (X_test, _) = mnist.load_data()
X_train = X_train.astype('float32') / 255.
X_test = X_test.astype('float32') / 255.
X_train = X_train.reshape((X_train.shape[0], -1))
X_test = X_test.reshape((X_test.shape[0], -1))
```

A numpy trick to flatten the rest of the dimension is to use -1 to infer the new dimension's size based on the old one. Since our input is 60000x28x28, using -1 for the last dimension, will

effectively flatten the rest of the dimensions. Hence, our resulting shape is 60000x784, for the training data.

Now we can create our autoencoder! We'll use ReLU neurons everywhere and create constants for our input size and our encoding size. Notice that our hidden layer size is much smaller than our input!

```
INPUT_SIZE = 784
ENCODING_SIZE = 64

input_img = Input(shape=(INPUT_SIZE,))
encoded = Dense(ENCODING_SIZE, activation='relu')(input_img)
decoded = Dense(INPUT_SIZE, activation='relu')(encoded)
autoencoder = Model(input_img, decoded)
```

Now we simply build and train our model. We'll use the ADAM optimizer and mean squared error loss (the Euclidean distance/loss) between the input and reconstruction.

```
autoencoder.compile(optimizer='adam', loss='mean_squared_error')
autoencoder.fit(X_train, X_train, epochs=50, batch_size=256, shuffle=True,
validation_split=0.2)
```

After our autoencoder has trained, we can try to encode and decode the test set to see how well our autoencoder can compress.

```
decoded_imgs = autoencoder.predict(X_test)
```

Finally, we can visualize our true values and reconstructions using matplotlib.

```
plt.figure(figsize=(20, 4))
for i in range(10):
    # original
    plt.subplot(2, 10, i + 1)
    plt.imshow(X_test[i].reshape(28, 28))
    plt.gray()
    plt.axis('off')

    # reconstruction
    plt.subplot(2, 10, i + 1 + 10)
```

```
    plt.imshow(decoded_imgs[i].reshape(28, 28))
    plt.gray()
    plt.axis('off')

plt.tight_layout()
plt.show()
```

We can see the results below. The top row is the input and the bottom row is the reconstruction.

That's not a bad reconstruction! By training only 50 epochs, we get decent reconstructions! If we wanted to be more rigorous about it, we could plot the loss functions of training and validation to ensure we had a low reconstruction loss, but, qualitatively, these look great!

Deep Autoencoders

We can apply the deep learning principle and use more hidden layers in our autoencoder to reduce and reconstruct our input. Our hidden layers have a symmetry where we keep reducing the dimensionality at each layer (the encoder) until we get to the encoding size, then, we expand back up, symmetrically, to the output size (the decoder).

Let's take a look at how we can modify our autoencoder to be a deep autoencoder. Making this change is fairly simple.

```
input_img = Input(shape=(INPUT_SIZE,))
encoded = Dense(512, activation='relu')(input_img)
encoded = Dense(256, activation='relu')(encoded)
encoded = Dense(128, activation='relu')(encoded)

encoded = Dense(ENCODING_SIZE, activation='relu')(encoded)

decoded = Dense(128, activation='relu')(encoded)
decoded = Dense(256, activation='relu')(decoded)
```

```
decoded = Dense(512, activation='relu')(decoded)
decoded = Dense(INPUT_SIZE, activation='relu')(decoded)
autoencoder = Model(input_img, decoded)
```

We reduce our input from 784 -> 512 -> 256 -> 128 -> 64, then expand it back up 64 -> 128 -> 256 -> 512 -> 784. The rest of the code is exactly the same.

These reconstructions look a bit better! If we plotted and compared the losses, this deeper autoencoder model actually has a smaller loss value than the shallow autoencoder model.

Convolutional Autoencoder

Remember that the MNIST data is *image data*. When we flatten it, we're not making full use of the spatial positioning of the pixels in the image. This is the same problem that plagued artificial neural networks when they were trying to work with image data. Instead, we developed Convolutional Neural Networks to handle image data.
(If you are not familiar with convolutional neural networks, please read the post here.)

Similarly, we can use **convolutional autoencoders**! Instead of using fully-connected layers, we use convolutional and pooling layers to reduce our input down to an encoded representation. But then we have to somehow upscale our encoded representation back up to the same shape as the encoding. How do we do that?

The opposite of convolution is (misleadingly named) *deconvolution*, but, in reality, this is just applying plain, old convolution! So to "undo" a convolution, we simply apply another convolution! I won't go into the details, but "undo-ing" a convolution corresponds to applying a transpose of a kernel, which is just convolution with a transposed kernel, which is just convolution! (A more apt name for *deconvolution* is *convolution transposed*.)

The opposite of pooling is upsampling. This works in the exact same way as pooling, but in reverse. For pooling, we split the image up into non-overlapping regions of a particular size and take the max of each region to become the new pixel (for max pooling). For example, if we had a pooling layer with a 2x2 window size, then each 2x2 window in the input corresponds to a single pixel in the output. For upsampling, we reverse this. For each pixel in the input, we expand it out to encompass a window. If we had a 2x2 upsampling window, then each pixel in

the input is resized to a 2x2 region. There are fancier ways we can do this, e.g., fancy interpolation, but upsampling works well in practice.

When coding a convolutional autoencoder, we have to make sure our input has the correct shape. The MNIST data we get will be only 28x28, but we also expect a dimension that tells us the number of channels. MNIST is in black-and-white, so we only have a single channel. We can easily extend our data by a dimension using numpy's newaxis. The ellipsis before the newaxis just means to leave everything else as is. This effectively changes our shape from 28x28 to 28x28x1 (and the batch size is included as the first dimension).

```
from keras.layers import Input, Dense, Conv2D, MaxPooling2D, UpSampling2D
from keras.models import Sequential
from keras.datasets import mnist

import numpy as np
import matplotlib.pyplot as plt

(X_train, _), (X_test, _) = mnist.load_data()
X_train = X_train.astype('float32') / 255.
X_test = X_test.astype('float32') / 255.
X_train = X_train[..., np.newaxis]
X_test = X_test[..., np.newaxis]
```

Now we can build our convolutional autoencoder! Notice that we're using a slightly different syntax. Instead of using Keras' functional notation, we're using a sequential model where we simply add layers to it in the order we want. The functional syntax is for more advanced models where we might several inputs and several output or other non-sequential models. We could have represented the previous deep and shallow autoencoders with sequential models as well.

```
autoencoder = Sequential()
autoencoder.add(Conv2D(32, (3, 3), activation='relu', padding='same',
input_shape=(28, 28, 1)))
autoencoder.add(MaxPooling2D((2, 2), padding='same'))
autoencoder.add(Conv2D(8, (3, 3), activation='relu', padding='same'))
autoencoder.add(MaxPooling2D((2, 2), padding='same'))
autoencoder.add(Conv2D(8, (3, 3), activation='relu', padding='same'))

# our encoding
```

```
autoencoder.add(MaxPooling2D((2, 2), padding='same'))

autoencoder.add(Conv2D(8, (3, 3), activation='relu', padding='same'))
autoencoder.add(UpSampling2D((2, 2)))
autoencoder.add(Conv2D(8, (3, 3), activation='relu', padding='same'))
autoencoder.add(UpSampling2D((2, 2)))
autoencoder.add(Conv2D(32, (3, 3), activation='relu'))
autoencoder.add(UpSampling2D((2, 2)))
autoencoder.add(Conv2D(1, (3, 3), activation='relu', padding='same'))
```

For each convolutional layer, we have a corresponding "deconvolutional" layer when we decode. For each max pooling layer, we have a corresponding upsampling layer. After we've constructed this architecture, we can simply build and fit our model.

```
autoencoder.compile(optimizer='adam', loss='mean_squared_error')
autoencoder.fit(X_train, X_train, epochs=50, batch_size=256, shuffle=True,
validation_split=0.2)
```

Let's take a look at the results of the convolutional autoencoder.

These results are much smoother! Notice that we've filled in some of the holes/gaps in our numbers. Just like how convolutional neural networks performed better than plain artificial neural networks on image tasks, convolutional autoencoders work better than plain autoencoders.

Denoising Autoencoder

One application of convolutional autoencoders is denoising. Suppose we have an input image with some noise. These kinds of noisy images are actually quite common in real-world scenarios. For a **denoising autoencoder**, the model that we use is identical to the convolutional autoencoder. However, our training and testing data are different. For our training data, we add random, Gaussian noise, and our test data is the original, clean image. This trains our denoising autoencoder to produce clean images given noisy images.

We'll have to add noise to our training data. This is simple enough with numpy. We also have to clip our image pixels to $[0, 1]$.

```
(X_train, _), (X_test, _) = mnist.load_data()
X_train = X_train.astype('float32') / 255.
X_test = X_test.astype('float32') / 255.
X_train = X_train[..., np.newaxis]
X_test = X_test[..., np.newaxis]

X_train_noisy = X_train + 0.25 * np.random.normal(size=X_train.shape)
X_test_noisy = X_test + 0.25 * np.random.normal(size=X_test.shape)

X_train_noisy = np.clip(X_train_noisy, 0., 1.)
X_test_noisy = np.clip(X_test_noisy, 0., 1.)
```

We have to make some modifications to how we use our new, noisy data. When we fit our model, the input is the noisy data and the output is the clean data. Everything else is the same as the convolutional autoencoder.

```
autoencoder.fit(X_train_noisy, X_train, epochs=50, batch_size=256,
shuffle=True, validation_split=0.2)
```

Now, we can plot the the noisy test examples and the denoised examples.

```
decoded_imgs = autoencoder.predict(X_test_noisy)

plt.figure(figsize=(20, 4))
for i in range(10):
    # original
    plt.subplot(2, 10, i + 1)
    plt.imshow(X_test_noisy[i].reshape(28, 28))
    plt.gray()
    plt.axis('off')

    # reconstruction
    plt.subplot(2, 10, i + 1 + 10)
    plt.imshow(decoded_imgs[i].reshape(28, 28))
    plt.gray()
    plt.axis('off')

plt.tight_layout()
```

```
plt.show()
```

Let's see if our denoising autoencoder works.

We've successfully cleaned up our images! Given new images, we can apply our autoencoder to remove noise from our images. Remember that autoencoders are only as good as the data we train them on. Since we trained our autoencoder on MNIST data, it'll only work for MNIST-like data!

To summarize, an autoencoder is an *unsupervised neural network* comprised of an encoder and decoder that can be used to compress the input data into a smaller representation and uncompress it. They don't have to be 2-layer networks; we can have deep autoencoders where we symmetrically stack the encoder and decoder layers. Additionally, when dealing with images, autoencoders don't have to use just fully-connected layers! We can use convolution and pooling, and their counterparts, to encode and decode image data! In fact, we noticed that we actually get better-looking reconstructions. One application of these convolutional autoencoders that we discussed was for image denoising. We can train an autoencoder to clean up noisy images and found they worked quite well!

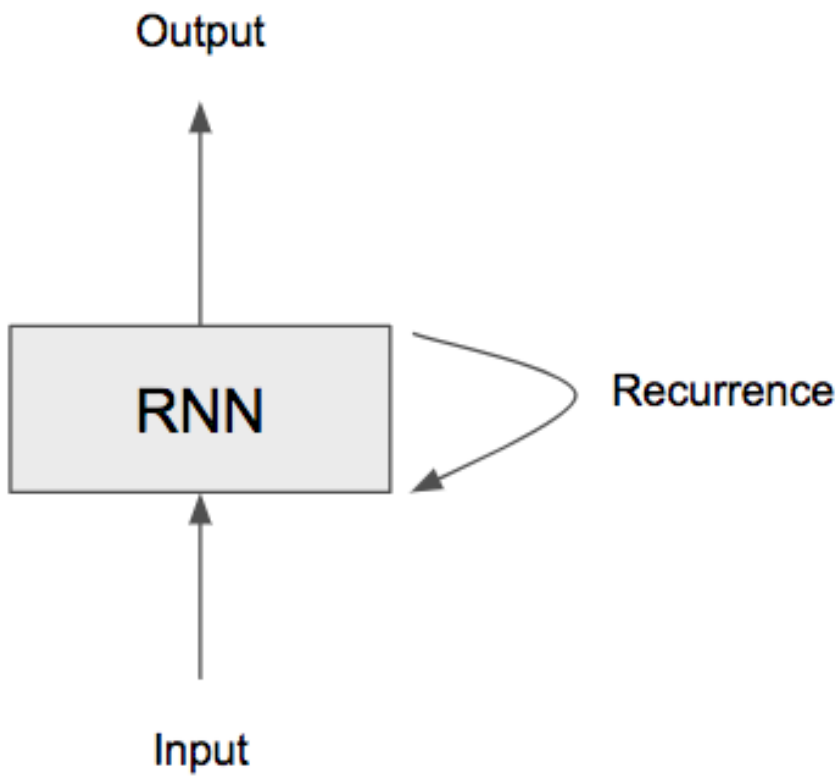
Autoencoders are one of the simplest unsupervised neural networks that have a wide variety of use in compression.

Recurrent Neural Networks for Language Modeling

Many neural network models, such as plain artificial neural networks or convolutional neural networks, perform really well on a wide range of data sets. They're being used in mathematics, physics, medicine, biology, zoology, finance, and many other fields. However, there is one major flaw: they require fixed-size inputs! The inputs to a plain neural network or convolutional neural network have to be the same size for training, testing, and deployment! This means we can't use these architectures for sequences or time-series data.

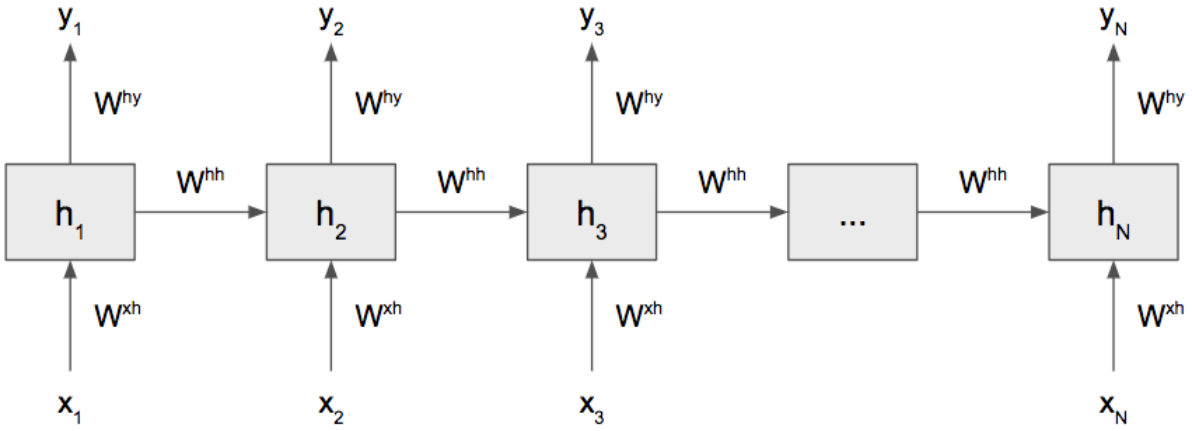
But along comes **recurrent neural networks** to save the day! We'll define and formulate recurrent neural networks (RNNs). We'll discuss how we can use them for sequence modeling as well as **sequence generation**. Then we'll code up a generic, character-based recurrent neural network from scratch, without any external libraries besides numpy! Finally, we'll train our RNN on Shakespeare and have it generate new Shakespearean text!

Recurrent Neural Networks Formulation



As we mentioned before, recurrent neural networks can be used for modeling variable-length data. The most general and fundamental RNN is shown above. The most important facet of the RNN is the **recurrence**! By having a loop on the internal state, also called the **hidden state**, we can keep looping for as long as there are inputs.

The above image can be a bit difficult to understand in practice, so we commonly "unroll" the RNN where we have a box for each **time step**, or input in the sequence.



For a particular cell, we feed in an input $\mathbf{x}_t$ at some time t to get a hidden state $\mathbf{h}_t$; then, we use that to produce an output $\mathbf{y}_t$. We keep doing this until we reach the end of the sequence. The most important part of an RNN is the recurrence, and that is modeled by the arrow that goes from one hidden state block to another. This recurrence indicates a dependence on all the information prior to a particular time t. In other words, inputs later in the sequence should depend on inputs that are earlier in the sequence; the sequence isn't independent at each time step!

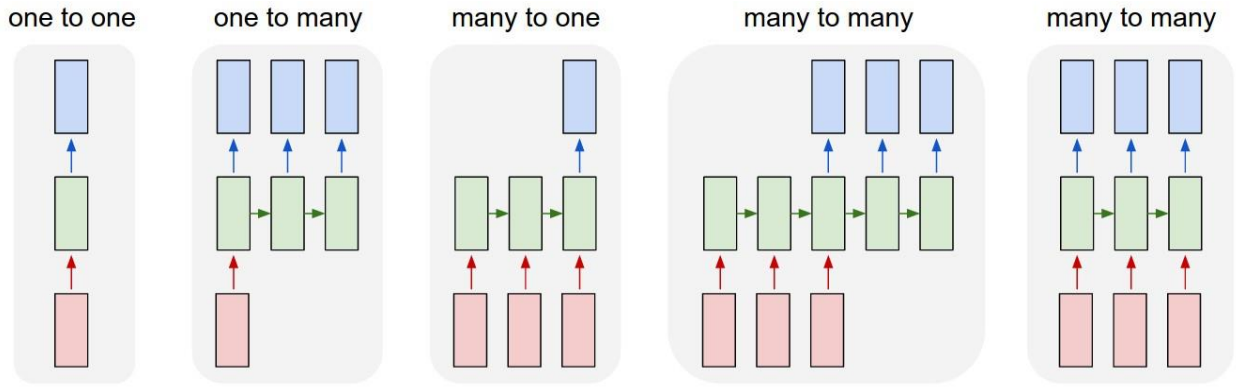
For example, suppose we were doing language modeling. We have an input sentence: "the cat sat on the ____." By knowing all of the words before the blank, we have an idea of what the blank should or should not be! This is the reason RNNs are used mostly for language modeling: they represent the sequential nature of language!

For our purposes, we're going to be coding a character-based RNN. This takes character input and produces character output. We take our text and split it into individual characters and feed that in as input. However, we can't directly feed text into our RNN. All neural networks work with numbers, not characters! However, we can easily convert characters to their numerical counterparts. We simply assign a number to each unique character that appears in our text; then we can convert each character to that number and have numerical

inputs! Similarly, our output will also be numerical, and we can use the inverse of that assignment to convert the numbers back into texts.
(In practice, when dealing with words, we use **word embeddings**, which convert each string word into a dense vector. Usually, these are trained jointly with our network, but there are many different pre-trained word embedding that we can use off-the-shelf (Richard Socher's pre-trained GloVe embeddings, for example).)

Like any neural network, we have a set of weights that we want to solve for using gradient descent: $\mathbf{W}^{(xh)}$, $\mathbf{W}^{(hh)}$, $\mathbf{W}^{(hy)}$ (I'm excluding the biases for now). Unlike other neural networks, these weights *are shared for each time step!* We use the same weights for each time step! This is also part of the *recurrence* aspect of our RNN: the weights are affected by the entire sequence. This makes training them a bit tricky, as we'll discuss soon.

The above figure models an RNN as producing an output at each time step; however, this need not be the case. We can vary how many inputs and outputs we have, as well as when we produce those outputs.



(Credit: http://karpathy.github.io/2015/05/21/rnn-effectiveness/)

We can have several different flavors of RNNs:

1. One-to-one: the simplest RNN
2. One-to-many: sequence generation tasks
3. Many-to-one: sentiment analysis
4. Many-to-many: machine translation

Additionally, we can have bidirectional RNNs that feed in the input sequence in both directions! We can also stack these RNNs in layers to make deep RNNs. RNNs are just the basic, fundamental model for sequences, and we can always build upon them.

Backpropagation Through Time

Now that we understand the intuition behind an RNN, let's formalize the network and think about how we can train it. For our purposes, we're just going to consider a very simple

RNN, although there are more complicated models, such as the long short-term memory (LSTM) cell and gated recurrent unit (GRU).
An RNN is essentially governed by 2 equations.

$$\mathbf{h}_t = \sigma(\mathbf{W}^{(xh)}\mathbf{x}_t + \mathbf{W}^{(hh)}\mathbf{h}_{t-1} + \mathbf{b}^{(h)})$$
$$\mathbf{y}_t = \mathbf{W}^{(hy)}\mathbf{h}_t + \mathbf{b}^{(y)} \qquad (1)$$

The first defines the recurrence relation: the hidden state at time t is a function of the input at time t and the previous hidden state at time $t-1$. For $\mathbf{h}_0$, we usually initialize that to the zero vector. For our nonlinearity, we usually choose **hyperbolic tangent** or **tanh**, which looks just like a sigmoid, except it is between -1 and 1 instead of 0 and 1. The second equation simply defines how we produce our output vector. (It looks almost exactly like a single layer in a plain neural network!)

Speaking of vectors, notice that everything in our RNN is essentially a vector or matrix. Our input and output dimensionality are determined by our data. However, we choose the size of our hidden states! We'll discuss more about the inputs and outputs when we code our RNN. Notice that we have a total of 5 parameters: $\mathbf{W}^{(xh)}$, $\mathbf{W}^{(hh)}$, $\mathbf{b}^{(h)}$, $\mathbf{W}^{(hy)}$, $\mathbf{b}^{(y)}$. We need to come up with update rules for each of these equations. We won't derive the equations, but let's consider some challenges in applying backpropagation for sequence data.

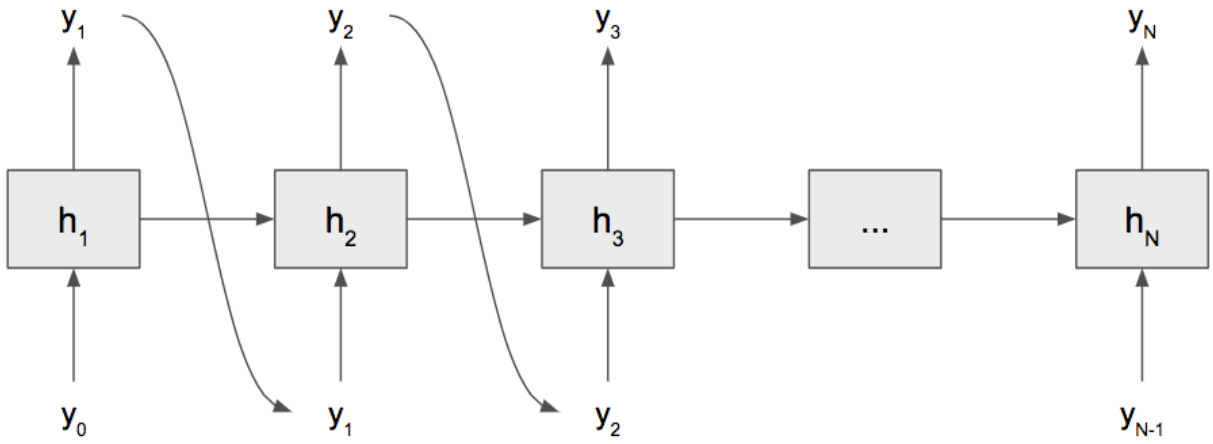
We call this kind of backpropagation, **backpropagation through time**. We essentially unroll our RNN for some fixed number of time steps and apply backpropagation. However, we have to consider the fact that we're applying the error function *at each time step!* So our total error is simply the sum of all of the errors at each time step. This is different than backpropagation with plain neural networks because we only apply the cost function *once* at the end.

The most difficult component of backpropagation through time is how we compute the hidden-to-hidden weights $\mathbf{W}^{(hh)}$. Although we can use the chain rule, we have to be very careful because we're using the same $\mathbf{W}^{(hh)}$ for each time step! At a particular time t, the hidden state depends on all previous time steps. We have to add up *each contribution* when computing this matrix of weights. In other words, we have to backpropagate the gradients from t back to all time steps before t. Like backpropagation for regular neural networks, it is easier to define a δ that we pass back through the time steps.

RNNs for Sequence Generation

After our RNN is trained, we can use it to generate new text based on what we've trained it on! For example, if we trained our RNN on Shakespeare, we can generate new Shakespearean text! There are several different ways of doing this (beam search is the most popular), but we're going to use the simplest technique called **ancestral sampling**. The idea is

to create a probability distribution over all possible outputs, then randomly sample from that distribution. Then that sample becomes the input to the next time step, and we repeat for however long we want.



We need to pick the first character, called the **seed**, to start the sequence. Then, using ancestral sampling, we can generate arbitrary-length sequences! (The reason this is called ancestral sampling is because, for a particular time step, we condition on all of the inputs before that time step, i.e., its ancestors.)

In the specific case of our character model, we seed with an arbitrary character, and our model will produce a probability distribution over all characters as output. This probability distribution represents which of the characters in our corpus are most likely to appear next. Then we randomly sample from this distribution and feed in that sample as the next time step. Repeat until we get a character sequence however long we want!

But how do we create a probability distribution over the output? We can use the **softmax function!** Our output is essentially a vector of scores that is as long as the number of words/characters in our corpus.

$$softmax(\mathbf{y})_k = \frac{e^{\mathbf{y}_k}}{\sum_{j=1}^{M} e^{\mathbf{y}_j}}$$

Above, suppose our output vector has a size of M. This function simply selects each component of the vector $\mathbf{y}$, takes e to the power of that component, and sums all of those up to get the denominator (a scalar). Then, we divide each component of $\mathbf{y}$ by that sum. The output is a probability distribution over all possible words/characters! Then we can sample from this distribution!

Coding an RNN

Now that we have an intuitive, theoretical understanding of RNNs, we can build an RNN! We're going to build a character-based RNN (CharRNN) that takes a text, or corpus, and learns character-level sequences. We can use that same, trained RNN to generate text.

Let's get started by creating a class and initializing all of our parameters, hyperparameters, and variables.

```
class CharRNN(object):
    def __init__(self, corpus, hidden_size=128, seq_len=25, lr=1e-3,
epochs=100):
        self.corpus = corpus
        self.hidden_size = hidden_size
        self.seq_len = seq_len
        self.lr = lr
        self.epochs = epochs

        chars = list(set(corpus))
        self.data_size, self.input_size, self.output_size = len(corpus),
len(chars), len(chars)
        self.char_to_num = {c:i for i,c in enumerate(chars)}
        self.num_to_char = {i:c for i,c in enumerate(chars)}

        self.h = np.zeros((self.hidden_size , 1))

        self.W_xh = np.random.randn(self.hidden_size, self.input_size) *
0.01
        self.W_hh = np.random.randn(self.hidden_size, self.hidden_size) *
0.01
        self.W_hy = np.random.randn(self.output_size, self.hidden_size) *
0.01
        self.b_h = np.zeros((self.hidden_size, 1))
        self.b_y = np.zeros((self.output_size, 1))
```

The corpus is the actual text input. We then create lookup dictionaries to convert from a character to a number and back. Finally, we initialize all of our weights to small, random noise and our biases to zero. Notice we also initialize our hidden state to the zero vector.

Speaking of sampling, let's write the code to sample. Remember that we need an initial character to start with and the number of characters to generate.

```
def sample(self, seed, n):
```

```
    seq = []
    h = self.h

    x = np.zeros((self.input_size, 1))
    x[self.char_to_num[seed]] = 1

    for t in range(n):
        # forward pass
        h = np.tanh(np.dot(self.W_xh, x) + np.dot(self.W_hh, h) + self.b_h)
        y = np.dot(self.W_hy, h) + self.b_y
        p = np.exp(y) / np.sum(np.exp(y))

        # sample from the distribution
        seq_t = np.random.choice(range(self.input_size), p=p.ravel())

        x = np.zeros((self.input_size, 1))
        x[seq_t] = 1
        seq.append(seq_t)
    return ''.join(self.num_to_char[num] for num in seq)
```

Let's suppose that all of our parameters are trained already. For a given number of time steps, we do a forward pass of the current input and create a probability distribution over the next character using softmax. Then, we randomly sample from that distribution to become our input for the next time step. We're also recording the number so we can re-map it to a character when we print it out.

To clean up the code and help with understanding, we're going to separate the code that trains our model from the code that computes the gradients. First, we'll define the function to train our model since it's simpler and help abstract the gradient computations.

```
def fit(self):
    smoothed_loss = -np.log(1. / self.input_size) * self.seq_len
    for e in range(self.epochs):
        for p in range(np.floor(self.data_size /
self.seq_len).astype(np.int64)):
            # get a slice of data with length at most seq_len
            x = [self.char_to_num[c] for c in self.corpus[p *
self.seq_len:(p + 1) * self.seq_len]]
            y = [self.char_to_num[c] for c in self.corpus[p * self.seq_len
+ 1:(p + 1) * self.seq_len + 1]]
```

```
            # compute loss and gradients
            loss, dW_xh, dW_hh, dW_hy, db_h, db_y = self.__loss(x, y)
            smoothed_loss = smoothed_loss * 0.99 + loss * 0.01
            if p % 1000 == 0: print('Epoch {0}, Iter {1}: Loss:
{2:.4f}'.format(e+1, p, smoothed_loss))

            # SGD update
            for param, dparam in zip([self.W_xh, self.W_hh, self.W_hy,
self.b_h, self.b_y], [dW_xh, dW_hh, dW_hy, db_h, db_y]):
                param += -self.lr * dparam
```

We smooth our loss so it doesn't appear to be jumping around, which loss tends to do. The outermost loop simply ensures we iterate through all of the epochs. The inner loop actually splits our entire text input into chunks of our maximum sequence length. Then we convert each character into a number using our lookup dictionary. Notice that our outputs are just the inputs shifted forward by one character. It may look like we're doing unsupervised learning, but *RNNs are supervised learning models*! We use a function to compute the loss and gradients. We report the smoothed loss and epoch/iteration as well. Finally, with the gradients, we can perform a gradient descent update.

Now all that's left to do is compute the loss and gradients for a given sequence of text. Like any neural network, we do a forward pass and use backpropagation to compute the gradients.

```
def __loss(self, X, Y):
    xs, hs, ys, ps = {}, {}, {}, {}
    hs[-1] = np.copy(self.h)

    # forward pass
    loss = 0
    for t in range(len(X)):
        xs[t] = np.zeros((self.input_size, 1))
        xs[t][X[t]] = 1
        hs[t] = np.tanh(np.dot(self.W_xh, xs[t]) + np.dot(self.W_hh, hs[t-
1]) + self.b_h)
        ys[t] = np.dot(self.W_hy, hs[t]) + self.b_y
        ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t]))
        loss += -np.log(ps[t][Y[t], 0])
```

```
    # backward pass
    dW_xh = np.zeros_like(self.W_xh)
    dW_hh = np.zeros_like(self.W_hh)
    dW_hy = np.zeros_like(self.W_hy)
    db_h = np.zeros_like(self.b_h)
    db_y = np.zeros_like(self.b_y)
    delta = np.zeros_like(hs[0])
    for t in reversed(range(len(X))):
        dy = np.copy(ps[t])
        # backprop into y
        dy[Y[t]] -= 1
        dW_hy += np.dot(dy, hs[t].T)
        db_y += dy

        # backprop into h
        dh = np.dot(self.W_hy.T, dy) + delta
        dh_raw = (1 - hs[t] ** 2) * dh
        db_h += dh_raw
        dW_hh += np.dot(dh_raw, hs[t-1].T)
        dW_xh += np.dot(dh_raw, xs[t].T)

        # update delta
        delta = np.dot(self.W_hh.T, dh_raw)
    for dparam in [dW_xh, dW_hh, dW_hy, db_h, db_y]:
        # gradient clipping to prevent exploding gradient
        np.clip(dparam, -5, 5, out=dparam)

    # update last hidden state for sampling
    self.h = hs[len(X) - 1]
    return loss, dW_xh, dW_hh, dW_hy, db_h, db_y
```

The first loop simply computes the forward pass. The next loop computes all of the gradients. We didn't derive the backpropagation rules for an RNN since they're a bit tricky, but they're written in code above. (We use the **cross-entropy cost function**, which works well for categorical data.) Additionally, we perform **gradient clipping** due to the exploding gradient problem.

The **exploding gradient problem** occurs because of how we compute backpropagation: we multiply many partial derivatives togethers. In a long product, if each term

is greater than 1, then we keep multiplying large numbers together and can overflow! So we clip the gradient. Similarly, we can encounter the **vanishing gradient problem** if those terms are less than 1. Multiplying many numbers less than 1 produces a gradient that's almost zero! There are more advanced and complicated RNNs that can handle vanishing gradient better than the plain RNN.

That's all the code we need! Now we can start using it on any text corpus!

```
if __name__ == '__main__':
    with open('input.txt', 'r') as f:
        data = f.read()

    char_rnn = CharRNN(data, epochs=10)
    char_rnn.fit()
    print(char_rnn.sample(data[0], 100))
```

In the ZIP file, there's a corpus of Shakespeare that we can train on and generate Shakespearean text! (The code we wrote is not optimized, so training may be slow!) Below are some examples of Shakespearean text that the RNN may produce!

SEBASTIANHI:
It is flet'st and telefunture of Muckity heed.

ANTONIO:
What? Oo it is the stish is eyemy
Sseemaineds, let thou the, not spools would of
Noveit!--

ANTONIO:
Parly own ame.
Navised salied,,
It is thou may Fill flle of thee neven serally indeet asceeting wink's sabisy's corrious,
Why's, wlendy to?m.

SEBASTING:
O, speak deliaw
Thouss, for tryiely cown lest umperfions.

SEBASTIAN:
Yo af or the pauges condest I waped possitly.

SEBASTIAN:
Py himsent

Winker thy broke,
gnded when
I"stey-ingery,
My gromise!

ADNONCA:
Nothed, Goon hearts
sprair,;
And come, the Nattrevanies, as impentnems,
Who will more; senfed to make he she a kist
Is is?

ANBONIO:
&ly not friend 'rissed's poke,
To undmensen;
Nolly the purrone,!
Seo'st as inlo good nature your sweactes subour, you are you not of diem suepf thy fentle.
What you There rost e'elfuly you allsm, I-loves your goods your emper
it: seave seep. Teck:
These standly servilph allart.

ENTUSSIN:
Nay, I doce mine other we kindd, speak, yo
ure winl, funot
Show natuees
sheant-that sofe,
While y am threat,
she sills;
You more speak, you dru
diss, it or my longly didan and to eftiscite:
My good serand
she, for yie, sir,
Can yoursely? They

Try this with other kinds of text corpa and see how well the RNN can learn the underlying language model!

Recurrent Neural Networks are neural networks that are used for sequence tasks. The flaw of previous neural networks was that they required a fixed-size input, but RNNs can operate on variable-length input! They share their parameters across sequences and are internally defined by a recurrence relation. We formulated RNNs and discussed how to train

them. Finally, we wrote code for a generic character-based RNN, trained it on a Shakespeare corpus, and had it generate Shakespeare for us!

Recurrent Neural Networks are the state-of-the-art neural architecture for advanced language modeling tasks like machine translation, sentiment analysis, caption generation, and question-answering!

Advanced Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are used in all of the state-of-the-art language modeling tasks such as machine translation, document detection, sentiment analysis, and information extraction. Previously, we've only discussed the plain, vanilla recurrent neural network. We'll be discussing state-of-the-art models that are used by companies like Google, Amazon, and Microsoft for language tasks. We'll first discuss the issue with vanilla RNNs. Then we'll discuss some state-of-the-art RNN models, such as the long short-term memory (LSTM) and gated recurrent unit (GRU).

The Issue with Vanilla RNNs

Recall that a vanilla RNN looks like the above figure and is completely described by the following equations .

$$h_t = \sigma(W^{(xh)}x_t + W^{(hh)}h_{t-1} + b^{(h)})$$

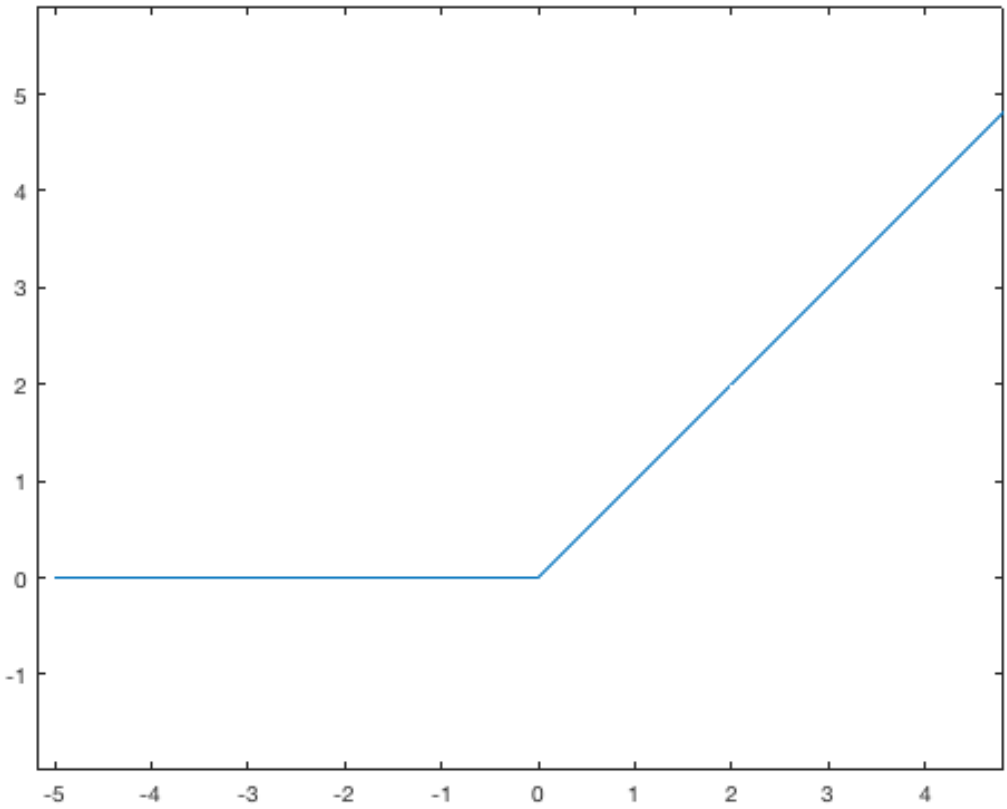
$$y_t = W^{(hy)}h_t + b^{(y)}$$

We can use the backpropagation-through-time algorithm to train our recurrent neural network. We use the chain rule of calculus to compute the partial derivative of the cost function with respect to the parameters and perform gradient descent to update them.

For longer sentences, we have to backpropagate through more time steps. When we do this, the gradient is multiplied by more and more terms. By the time we get to the first few layers, our gradient is a product of many terms! If many of these terms are very small, then the product, i.e., the gradient, will be close to zero. This is called the **vanishing gradient problem**. When our gradient is near zero, our parameters update very slowly.

In terms of language modeling, this means words that are far apart aren't considered when trying to predict the next word. The gradient is nearly zero when we reach the beginning of the sequence so it doesn't relay any information to the early words in the sequence.
One solution to the vanishing gradient is to use ReLU activations.

$$f(x) = \max(0, x)$$

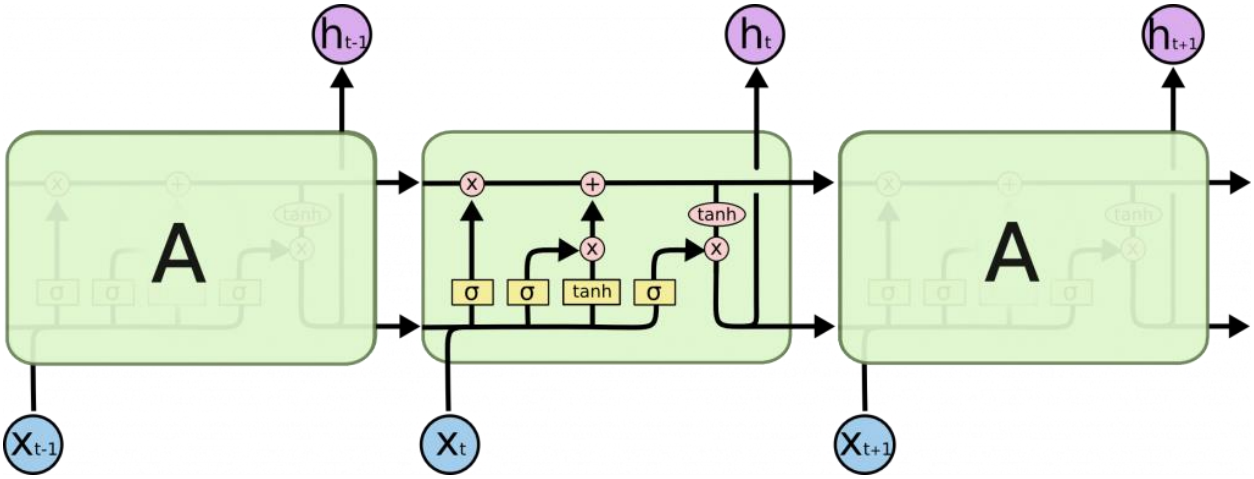


The derivative of the ReLU is 1 if the input is greater than 0 and 0 otherwise (technically, undefined at 0). So when we multiply these together in a long chain, we'll just be multiplying the gradient by 1 at each time step: it won't vanish!

On the other hand, if many of these terms are very large, then the product will be even larger! Large gradients are equally bad since we'll encounter overflow problems very quickly. This is called the **exploding gradient problem**. One quick solution is clipping the gradient at each step so that it is always between some values, e.g., between -5 and 5.

Long Short-Term Memory (LSTM)

The major issue with vanilla RNNs is the vanishing gradient problem, which prevents us from learning long-term dependencies. We can mitigate the effect by using ReLU activations, but that's still not quite enough. Instead, we can construct a new RNN cell that can learn these long-term dependencies. We call these **Long Short-Term Memory (LSTM)** networks. Here's a diagram of a single cell:



(Source: colah.github.io/posts/2015-08-Understanding-LSTMs/)

There's no way around it: these are complicated RNNs! But they produce much better results than plain RNNs so they're well worth the time spent understanding them.

There are more equations governing these networks than plain RNNs and they have a lot of moving parts, but we'll discuss each part and their effect on the overall RNN. The most important part of the LSTM is the **cell state**, the top horizontal line in the figure above. We can add/remove information to this state or clear it out based on the interactions of two gates: point-wise multiplication (also called the Hadamard product, denoted by $\odot$) and addition. These are the only two (linear) operations that modify this cell state so we can backpropagate effectively through these. Intuitively, this means we add or remove information to the cell state. This is different than the vanilla RNN, which only has a hidden state, no cell state.

The first thing we do is compute the **forget gate**: $f_t = \sigma(W_f \cdot [h_{t-1}; x_t] + b_f)$

We consider the previous hidden state and the input (concatenate) and compute an affine transform (multiply by W_f and add b_f) and pass it through a sigmoid. Due to the sigmoid, we produce an output between zero and one for each number in the cell state. So when we multiply this output with the cell state, we compute which elements we decide to keep (sigmoid output near one) and which to *forget* (sigmoid output near zero).

Consider an example sentence where we switch pronouns from "he" to "it". In this case, the forget gate should cause elements of the cell state to *forget* which noun the pronoun "he" refers to since it is no longer relevant.

The next step of the LSTM cell is to alter the cell state, through the addition gate. The first part of this is computing the **input gate**, which decides which elements of the cell state to update.

$$i_t = \sigma(W_i \cdot [h_{t-1}; x_t] + b_i)$$

This is the same kind of operation as f_t, except we have different weights W_i and biases b_i!

Next, we compute the actual values to be added to the cell state. i_t simply tells us which elements to update, but we have to compute the values that we will add.

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}; x_t] + b_C)$$

This is similar to previous two operations, but we use a hyperbolic tangent nonlinearity instead of a sigmoid to squash our values between -1 and 1. We don't use a sigmoid here is because sigmoid only produces values between 0 and 1, which prevents us from adding negative values to the cell state.

Now we can update the cell state using the following equation.

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$$

The first part of the sum applies the forget gate to the previous cell state. The other part of the sum applies the values of the update.

We have two final quantities to compute: the **output gate** and the hidden state. The **output gate** is similar to the input gate and forget gate.

$$o_t = \sigma(W_o \cdot [h_{t-1}; x_t] + b_o)$$

Note that this is different than the output of LSTM block, which is computed exactly the same as the vanilla RNN, i.e., some variant of $y_t = W_y h_t + b_y$. Finally we can compute the hidden state. We use the current, newly-updated cell state and the output gate to compute this.

$$h_t = o_t \odot \tanh(C_t)$$

The output gate is additional information that we fold into the hidden state. We can enumerate all of the equations that describe an LSTM.

$$\begin{aligned}
f_t &= \sigma(W_f \cdot [h_{t-1}; x_t] + b_f) \\
i_t &= \sigma(W_i \cdot [h_{t-1}; x_t] + b_i) \\
\tilde{C}_t &= \tanh(W_C \cdot [h_{t-1}; x_t] + b_C) \\
C_t &= f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \\
o_t &= \sigma(W_o \cdot [h_{t-1}; x_t] + b_o) \\
h_t &= o_t \odot \tanh(C_t)
\end{aligned}$$

LSTMs are complicated, but they produce state-of-the-art results for textual tasks. Let's use an LSTM to generate text, like what we did for the vanilla RNN.

LSTM Shakespeare Generation

In the ZIP file, I've included a corpus of snippets from Shakespeare's sonnets, and we'll be training an LSTM to generate new Shakespearean text. First, we need to read in our data and create dictionaries to convert between indices and characters.

```
from keras.models import Sequential
from keras.layers import Dense, LSTM, GRU
from keras.callbacks import Callback

import numpy as np
import random
import sys

with open('sonnets.txt') as f:
    text = f.read().lower()

chars = list(set(text))
char_to_ix = { ch:i for i, ch in enumerate(chars) }
ix_to_char = { i:ch for i, ch in enumerate(chars) }

data_size, vocab_size = len(text), len(chars)
print('Input size {0}, {1} unique'.format(data_size, vocab_size))

seq_len = 40

seqs_in = []
char_out = []
for i in range(0, data_size - seq_len):
    seqs_in.append(text[i:i+seq_len])
    char_out.append(text[i+seq_len])
```

Then we can chunk our data into sequences. Our goal is to predict the next character given a sequence of characters that comes before it. But we have to convert our data into numpy arrays so that Keras can use it.

```
input_size = len(seqs_in)

X = np.zeros((input_size, seq_len, vocab_size), dtype=np.bool)
y = np.zeros((input_size, vocab_size), dtype=np.bool)

for i, seq in enumerate(seqs_in):
```

```
    for j, char in enumerate(seq):
        X[i, j, char_to_ix[char]] = 1
    y[i, char_to_ix[char_out[i]]] = 1
```

We create an input tensor: input patterns, sequence length, and vocabulary size. For each input, we have a sequence of one-hot vectors, all the same length. For our output, we simply produce a one-hot vector of the next character.

We can create our model now: a single-layer LSTM with 128 hidden units, i.e., the dimensionality of the hidden state is 128.

```
model = Sequential()
model.add(LSTM(128, input_shape=(seq_len, vocab_size)))
model.add(Dense(vocab_size, activation='softmax'))

model.compile(loss='categorical_crossentropy', optimizer='adam')
print(model.summary())
```

Our LSTM takes in an input shape of a matrix: each row is a sequence and each column represents a one-hot vector. Remember that we omit the input size since Keras will automatically batch it for us.

We also have to write a function for sampling from our trained LSTM. We'll use ancestral sampling since it is the easiest to implement. Beam search tends to produce more coherent results, but is more complicated to implement.

```
def sample(preds):
    preds = np.asarray(preds).astype('float64')
    p = np.exp(preds) / np.sum(np.exp(preds))
    ix = np.random.choice(range(vocab_size), p=p.ravel())
    return ix
```

Since we're now using the LSTM as a black-box with Keras, we have to write a callback class that will sample the LSTM after each epoch.

```
class SampleCallback(Callback):
    def on_epoch_end(self, epoch, logs):
        start_index = random.randint(0, data_size - seq_len - 1)
        sentence = text[start_index: start_index + seq_len]
        print()
```

```
        print('Seed: ')
        print('-' * 16)
        print('"' + sentence + '"')
        print()
        print('Generated:')
        print('-' * 16)

        for i in range(200):
            x_pred = np.zeros((1, seq_len, vocab_size), dtype=np.bool)
            for k, char in enumerate(sentence):
                x_pred[0, k, char_to_ix[char]] = 1

            preds = self.model.predict(x_pred)
            next_ix = sample(preds)
            next_char = ix_to_char[next_ix]

            sentence = sentence[1:] + next_char

            sys.stdout.write(next_char)
            sys.stdout.flush()
        print('\n\n')
```

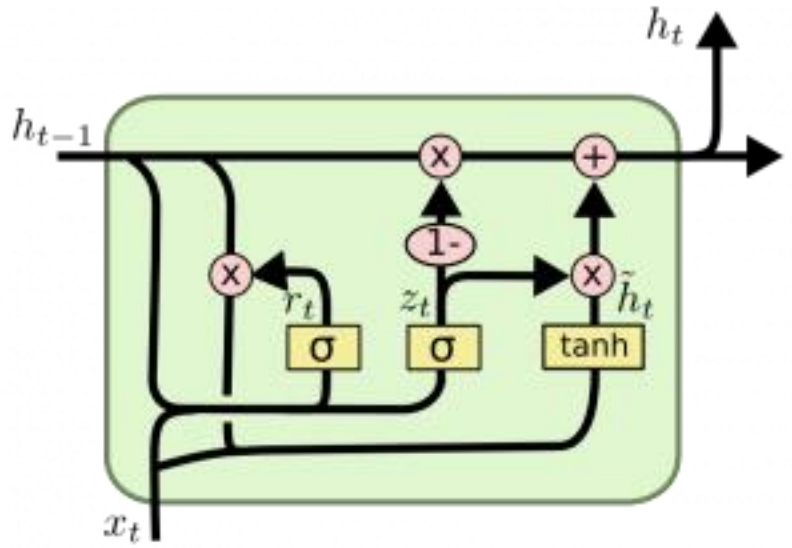
Here, we're generating 200 characters from the LSTM. We start with a seed sequence that is our initial input. Then we predict 200 characters by shifting the sequence by one character to maintain the same sequence length. Finally, we can train our model.

```
model.fit(X, y, batch_size=256, epochs=100, callbacks=[SampleCallback()])
```

Training LSTM is very difficult and time/resource-intensive. I highly recommend training this on an Nvidia GPU with CUDA rather than a CPU.

Gated Recurrent Unit (GRU)

The LSTM cell is complicated! There are a ton of different gates, but, actually, some of them serve similar purposes. We can take some of these gates and condense them to get a simpler cell: the **gated recurrent unit (GRU)**.



(Source: colah.github.io/posts/2015-08-Understanding-LSTMs/)
The equations that describe the GRU are simpler than the LSTM:

$$z_t = \sigma(W_z \cdot [h_{t-1}; x_t])$$
$$r_t = \sigma(W_r \cdot [h_{t-1}; x_t])$$
$$\tilde{h}_t = \tanh(W_h \cdot [r_t \odot h_{t-1}; x_t])$$
$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t$$

In the GRU, we can merge the input and forget gates into a single **update gate**. The intuition behind this is that we should make the decision to *forget* and *modify* together in one gate rather than two separate gates. That's where we get the equation for h_t. Additionally, cell state and hidden state seem to serve similar purposes, so let's also try to merge them together. Also notice that $\tilde{h}_t$ uses hyperbolic tangent, just like the information to add to the cell state! Since z_t returns values between 0 and 1, the equation for h_t merges the cell state and hidden state of the LSTM. If z_t is small, then $\tilde{h}_t$ won't have as much effect as the previous hidden state h_t. On the other hand, we can disregard the $\tilde{h}_t$ if z_t is large. This is like the input gate and update!

We also have r_t that is also between 0 and 1. We multiply this by the previous hidden state when computing $\tilde{h}_t$. If r_t is close to 0, then we zero-out most of the previous hidden state, acting like a forget gate!

But what about performance and quality? Many researchers have looked at these models and discovered they all perform very similarly. However, the GRU is quicker to train

because we have less computations to perform overall. We get similar quality when we sample from either.
Let's test this out! We can replace the LSTM with the GRU in our code very simply:

```
model.add(GRU(128, input_shape=(seq_len, vocab_size)))
```

Try training both of these models and see which is faster and produces the most coherent text!

To summarize, in this post, we discussed some of the issues that vanilla RNNs have. Although we can find a pseudo-fix to the vanishing gradient problem using ReLUs, we still have the problem of modeling long-term dependencies. To remedy this, we add more complexity to the RNN cell so that it can retain more information: a long short-term memory (LSTM) cell. We generated Shakespearean text using LSTMs. We also discussed a simpler variant of LSTMs called gated recurrent units (GRUs) that condense some of the LSTM gates. This leads to simpler computations and faster training.

LSTMs and GRUs are used in all of the state-of-the-art text processing tasks in research and at many large companies like Amazon, Google, and Microsoft!