CSCI E-25

Introduction to Deep Neural Networks

Steve Elston

1.0 Overview

This lesson introduces you to the basics of neural network architecture in the form of deep forward networks. This architecture is the quintessential deep neural net architecture. In this lesson you will master the following:

- Why is deep learning important and how it relates to representation, learning and inference.
- How a basic Preceptron works.
- How to apply different types of loss functions.
- Understand why nonlinear activation is important and why rectified linear units are a good choice.
- How back propagation works, and how you apply the chain rule of calculus to determine gradient.
- Understand the architectural trade-off between depth and width in deep networks.
- Know how and why you must apply regularization to deep neural networks.

1.1 Why is deep learning important?

Deep learning methods are a form of **artificial intelligence (AI)** or **machine intelligence**. More specifically, deep learning algorithms are a type of **machine learning**.

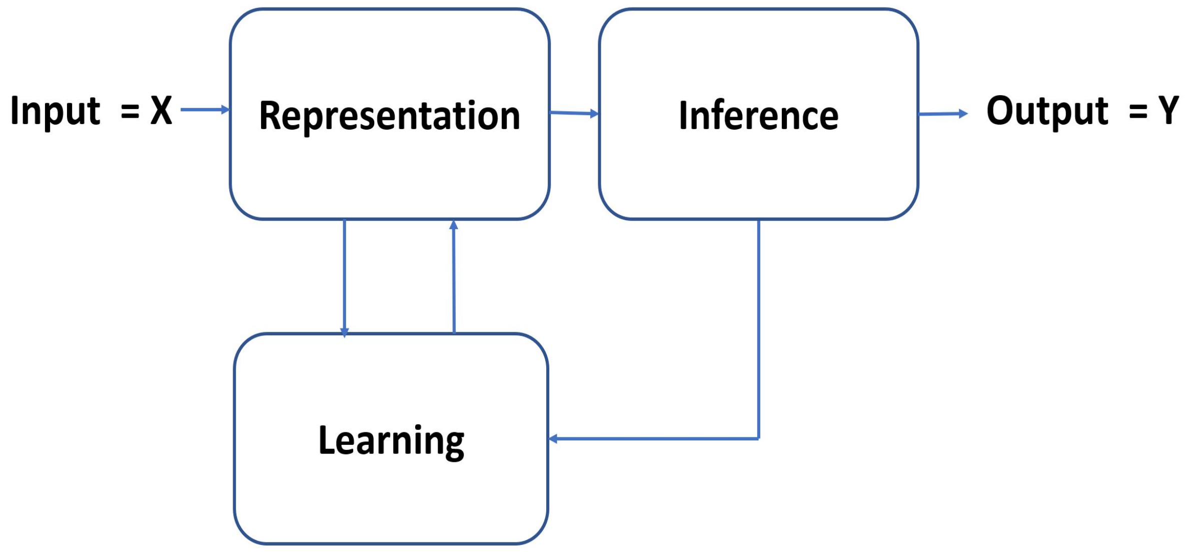
What properties does machine intelligence require? There have been many answers to this question over the history of computing. In this case, we will take a practical view, sometimes known as **weak AI**. There are three key properties an intelligent machine must have. Deep learning algorithms are one of a few classes of algorithms that can do the following, essential to machine intelligence:

1. **Representation:** An intelligent machine must be able to represent a model of the world it interacts with in a general manner. Representation is key to intelligence. Without a good representation the best learning and inference algorithms will struggle. Whereas, good representation can greatly facilitate learning and inference. In conventional machine learning the representation is model and a set of features. The representation is limited to what the features can provide directly. Deep learning algorithms, on the other hand, learn learn complex representations from raw features. This behavior allows

deep learning algorithms to approximate complex relationships. Further, the representations learned often generalize well, up to a point.

2. **Learning:** As you likely guessed from the very name, deep learning algorithms learn from data. Whereas, conventional machine learning is focused on inference,deep learning algorithms learn both inference and representations. As a result, deep leaning algorithms are more complex and therefore harder to train than conventional machine learning algorithms.
3. **Inference:** Any machine intelligence algorithm must be able to perform inference. The inference is the result produced given new input data. To be useful, the inferences produced by a machine intelligence algorithm must **generalize** beyond the cases used for learning or training. Good generalization requires both good representations and learning which can deal with the complexity of diverse situations. Some deep learning algorithms can approach human levels of performance in inference tasks such as recognizing objects in images or understanding natural speech.

The figure below shows a highly abstracted view of machine intelligence, showing the relationship between representation, learning and inference. In simple terms, the representation is learned and then used to make inferences. Errors in the inferences can be used to improve the learning of the representation.



Schematic for creating machine intelligence

That's it! The entire rest of this course will focus on just these three points: representation, learning and inference!

1.2 Installing Keras

This notebook will provides a first look at using the Keras package to define, train and evaluate deep learning models with Keras. The Keras package is a wrapper on TensorFlow, intended to abstract and simplify the definition, training and execution of TensorFlow deep learning models. You can find extensive well-written documentation for Keras here. The book Deep Learning with Python by François Chollet, the creator of Keras, provides in-depth examples and discussion on a wide range of deep learning applications.

By the end of this lesson you will be able to work with basic feedforward architecture multi-layer neural nets. Feedforward networks are one of a class of basic models called **sequential models** which are easy to define with Keras. Some basic regularization is introduced. Additional regularization methods are covered in a subsequent lesson.

Keras is part of the base package, as of the release of TensorFlow 2. Before proceeding make sure you have TensorFlow 2 installed in your environment. Follow these instructions..

Note: As an alternative to working with a local installation, you may choose to use the Google Colabratory. The Colabrotory virtual environment includes Anaconda, TensorFlow and Keras. However, the use of shared resources can result in slow execution.

Note: This notebook was constructed and tested using Anaconda 3 with Python 3. It is assumed that the standard Anaconda stack has been installed.

In [ ]:
```
from keras.datasets import mnist
import tensorflow as tf
import keras.utils.np_utils as ku
import keras.models as models
import keras.layers as layers
from keras import regularizers
import numpy as np
from math import exp
import matplotlib.pyplot as plt
import seaborn as sns

%matplotlib inline
```

2.0 Forward propagation: The representation problem

To create useful neutral network we need a **representation** that has two important properties.

First, there needs to be a way to represent complex functions of the input. Without this property, nothing is gained, since there are numerous machine learning algorithms that work with simple representations. We will spend the rest of this section exploring this problem.

Second, the representation needs to be learnable. Quite obviously, no machine intelligence representation is useful if there is not a practical algorithm to learn it. We will take up this problem in another section.

2.1 Linear networks

Let's start with the simplest possible network. It has inputs, and an output. The output is a **afine transformation** of the input values. We say this network performs an afine transformation since there is a bias term b.

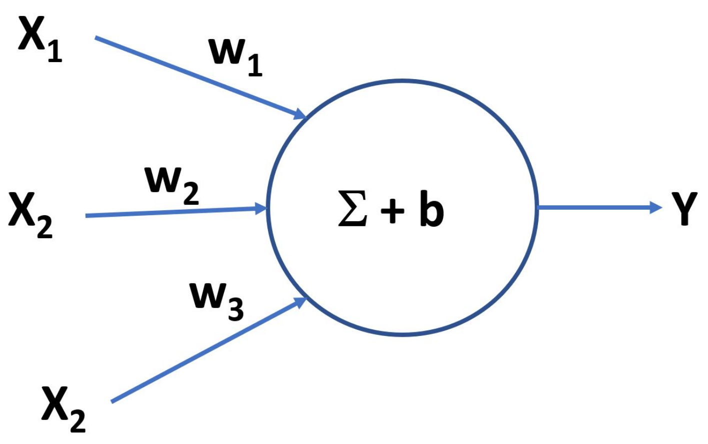


Figure 2.1 **A simple afine network**

This output y of this network is just:

$$y = f(x) = \sum_i w_i \cdot x_i + b$$

This network performs linear regression. Being able to perform only afine transformations, it can't do anything else.

This representation is certainly learnable. However, it does not gain us anything over familiar linear regression methods.

2.2 The preceptron

To get started, let's have a look at a simple **preceptron** model. The perceptron was proposed by Rosenblatt (1962). He built on the earlier attempts at a neural network models by McCulloch and Pitts (1943) and Heeb (1949). The perceptron adds **nonlinear activation** to the afine network.

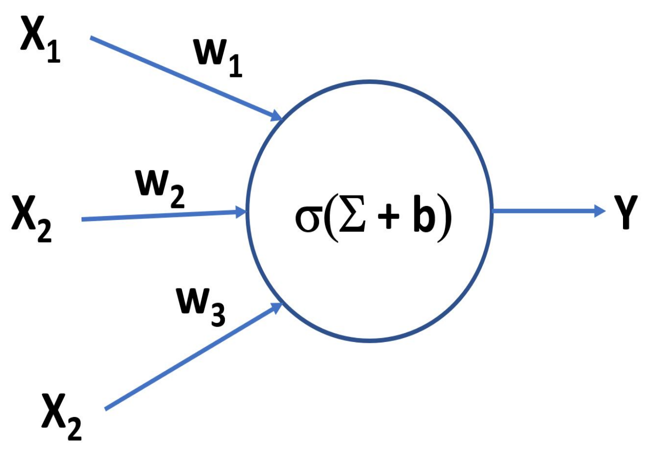


Figure 2.2 Schematic of perceptron with nonlinear activation

The output y of the perceptron is given by the following:

$$y = f(x) = \sigma\Big(\sum_i w_i \cdot x_i + b\Big)$$

The output of the network is now nonlinear, give the **activation function** $\sigma(x)$.

But, the preceptron is nothing more than a logistic regression classifier. The fact that the preceptron could only solve linearly separable problems was pointed out by Minsky and Papert (1969). The failure of the preceptron to learn an **exclusive or (XOR)** function is well known. See for example, Section 6.1 in GBC.

Again, this representation is certainly learnable. However, as before, it does not gain us anything over well known logistic regression models.

2.3 Forward networks - We're gonna need a better representation!

The problem with the perceptron is one of representations. There is no way that this simple network can represent anything but a linearly separable function. To represent more complex functions, we need a more complex network. In more technical terms we need a network with greater **model capacity**.

What we need is a network with layers of **hidden nodes**. The figure below shows a simple example of a neural network with one **hidden layer** with two nodes. Since every node (including inputs) is connected to every other node we call this architecture a **fully connected neural network**.

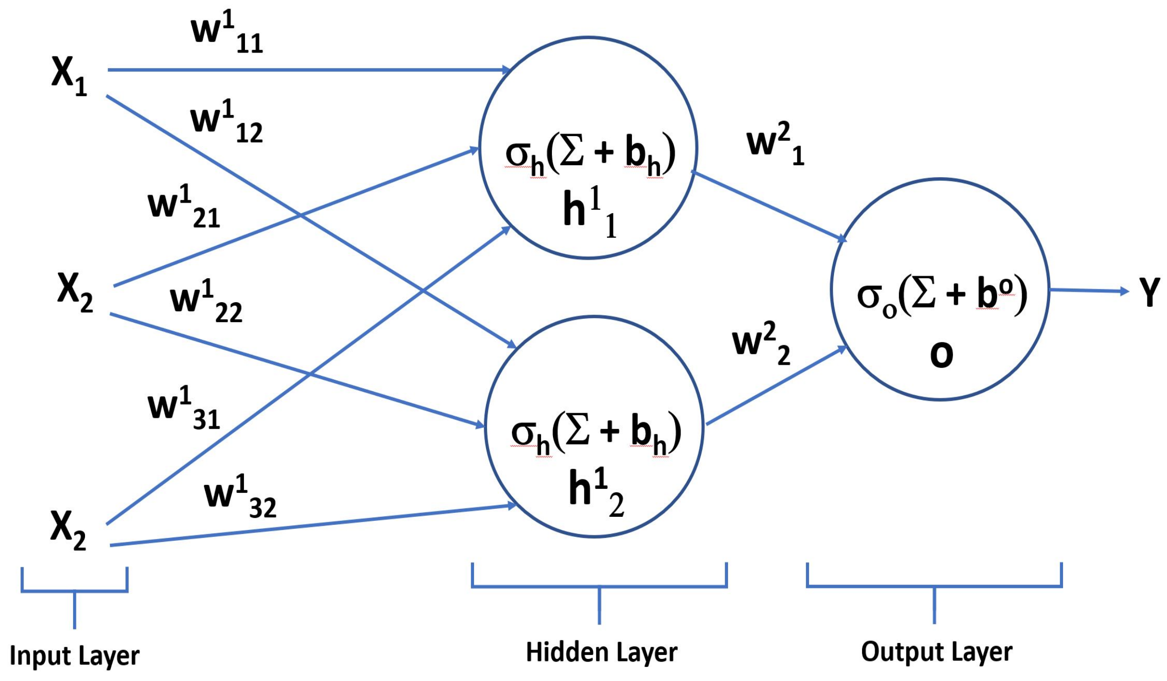


Figure 2.3 Fully connected neural network with single hidden layer

Let's walk through some aspects of these diagrams.

1. The neural network is divided into three layers. The input layer, the hidden layer and the output layer.
2. The values in the input layer are multiplied by a weight matrix, W^1.
3. The nodes in the hidden layer sum their inputs and add a bias term, b^1.
4. The outputs of the hidden layer nodes are multiplied by a weight vector, W^2.
5. The output layer sums the inputs and adds another bias term, b^2.

2.4 Neural network architectures - Finding representations

The representations achievable by neural network with just a single hidden layer are quite powerful. In fact, Cybenko (1989) showed that such a network with an infinite number of hidden units using sigmoidal activation can approximate any arbitrary function. Hornik (1991) generalized this to apply to any activation function. We call this theorem the **universal approximation theorem**.

A universal approximation theorem may see like a really exciting development; especially if you are a machine intelligence nerd. However, one must be circumspect when viewing such a result. A representation with an infinite number of nodes cannot be learned in any practical sense. Still it is comforting to know that, at least in principle, a representation can be learned for arbitrarily complex problems.

While infinitely wide networks with a single layer are unrealistic, we are not limited to one dimension. In fact, depth is typically more effective at creating complex representations rather than width in neural networks. Depth is measured by the count of hidden layers stacked one on top of the other in the network. Hence, the term deep neural networks.

The Figure 2.4 below shows the results of an empirical study by Goodfellow, Shlens and Szegedy (2014) of accuracy of the network vs depth. Notice that accuracy increases rapidly with depth until about 8 layers, after which the effect is reduced.

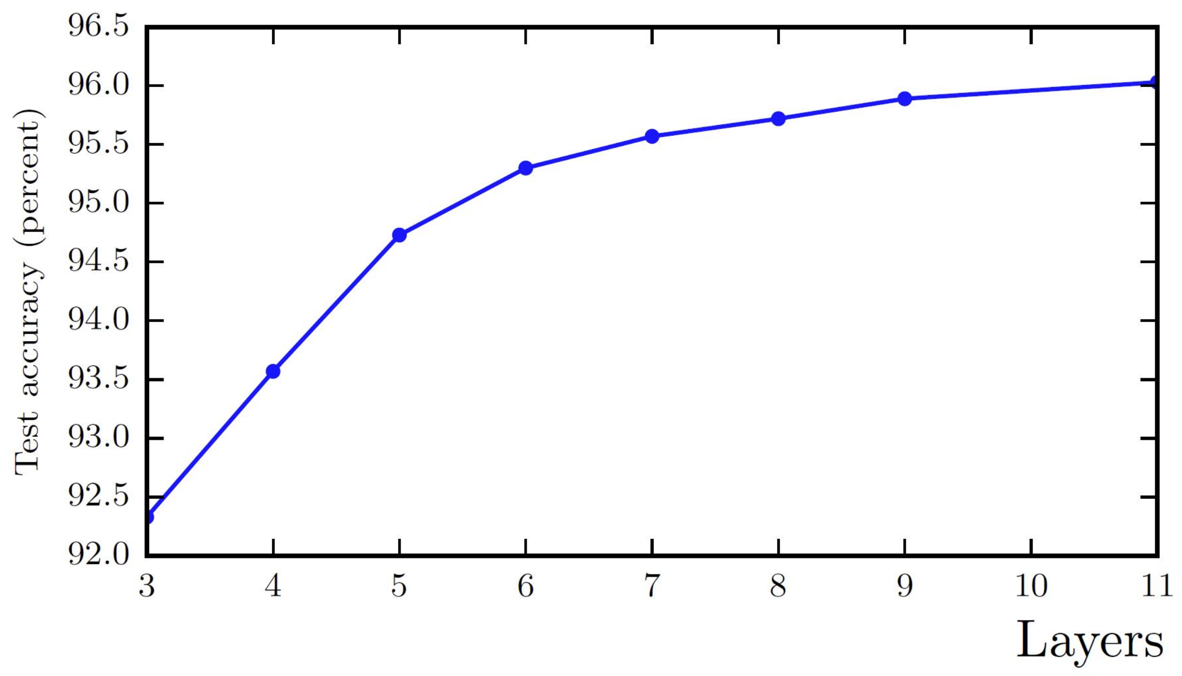


Figure 2.4 Empirical results of accuracy vs. number of layers Diagram from Goodfellow et. al. 2014

Another view of the empirical study by Goodfellow et. al. is shown in Figure 2.5 below. In this case accuracy verses number of model parameters is compared for three different network architectures. The deeper network (11 layers) makes more efficient use of the parameters in terms of improved accuracy. The number of parameters in a layer is approximately the total number of parameters divided by the number of layers. Notice that for the particular case tested convolutional neural networks are more efficient than fully-connected networks. We will discuss convolutional neural networks in a subsequent lesson.

Of particular interest is the fact that the fully-connected network and the shallow convolutional neural network appear to be over-fitting as the test accuracy actually decreases as the number of parameters increases. We will discuss the significant problems of over-fitting in neural networks in a subsequent lesson.

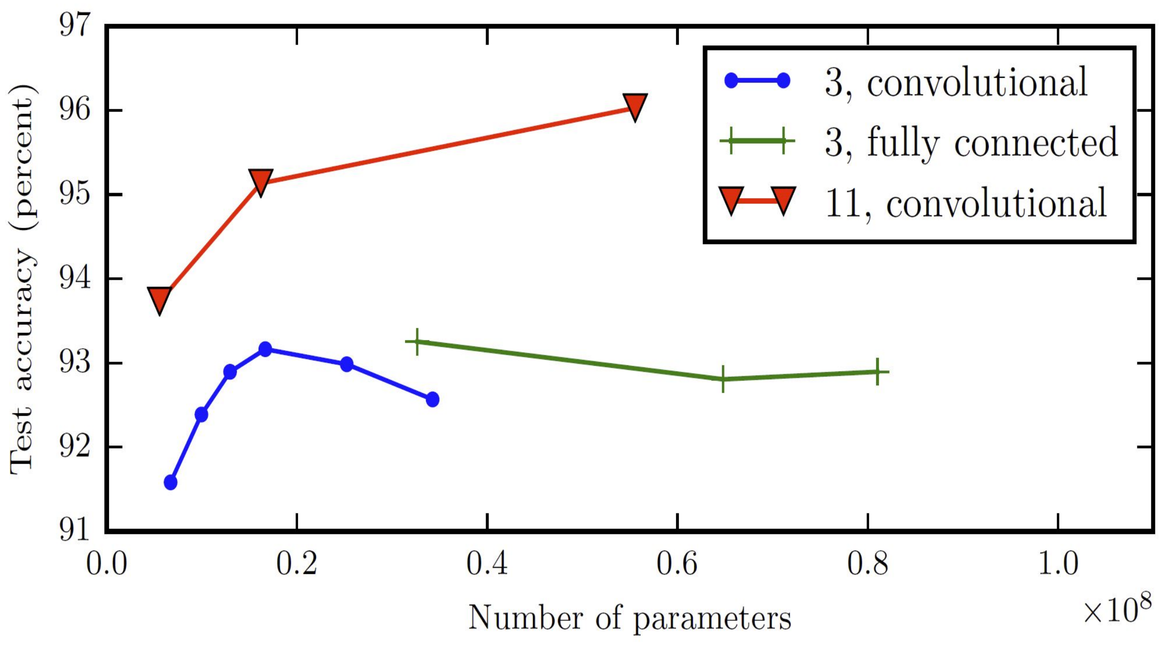


Figure 2.5 Empirical results of accuracy for different network architectures Diagram from Goodfellow et. al. 2014

Summary: Deep networks tend to produce better models, with less tendency to over-fit, for a given level of complexity.

2.5 Computational graphs

There is another way to look at neural nets, computational graphs. A computational graph breaks down the steps of a complex algorithm into steps.

Computational graphs provide a way to organize complex computations in an efficient manner. Widely used computational frameworks such as Tensor Flow, CNTK, and Torch all use computational graphs. Organizing computations in a graph allows these platform to minimize memory transfers. In simple terms, the platform can look ahead in the graph and organize data and computational results so as to minimize memory transfers. As a result, such platforms can be significantly faster than, say, Python Numpy. Systems like Numpy require memory transfer before each operation, which typically take more time than the actual computation.

The diagram below decomposes the single hidden layer neural network discussed in the previous section into a computational graph.

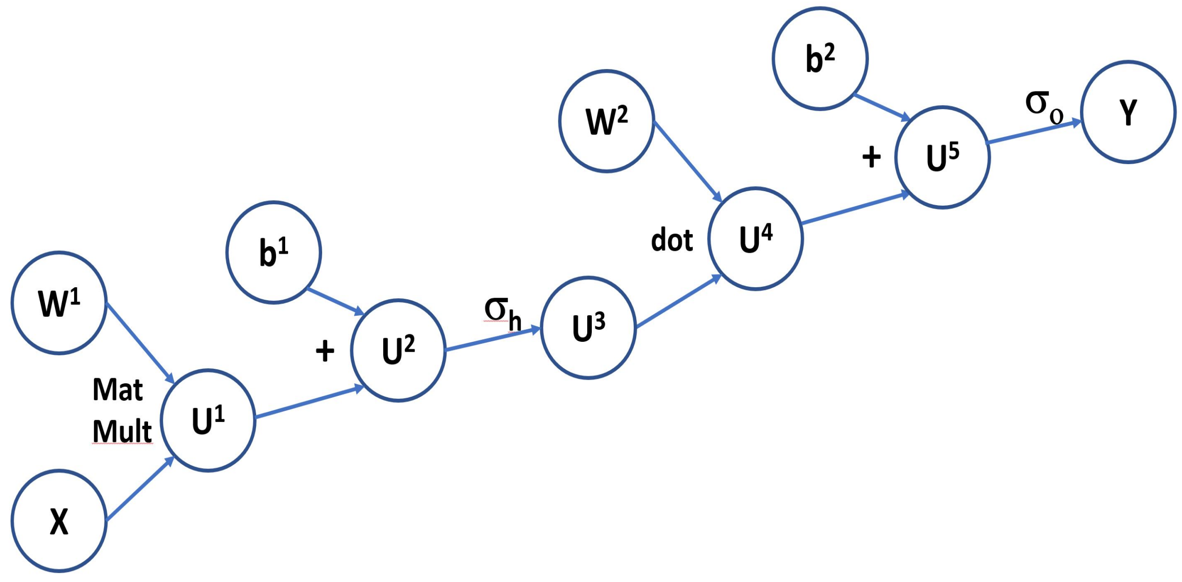


Figure 2.6 Computational graph for fully connected neural network of Figure 2.3
Let's walk though this graph, step by step.

1. The $nX2$ weight tensor, W^1, is multiplied by the 1-dimensional input tensor $x \in \ R^n$, giving the result $U^1 \in \ R^2$.
2. The 1-dimensional bias tensor $b^1 \in \ R^2$ is added to U^1, giving $U^2 \in \ R^2$.
3. The activation function $\sigma_h(x)$ is applied to U^2, producing $U^3 \in \ R^2$
4. The dot product between the weight tensor, $W^2 \in \ R^2$ and U^3 is computed giving $U^4 \in \ R^1$.
5. The bias, $b^2 \in \ R^1$ is added to U^4 giving $U^5 \in \ R^1$.
6. The output activation function $\sigma_o(x)$ is applied to U^4 giving the output $Y \in \ R^1$.

As you can see, the computational graph provides a complete specification for the single hidden layer neural network.

2.6 Activation functions

Without a nonlinear activation function, a neural net is just an afine transformation. Afine transformations limit representation to only linearly separable functions. To create more general representations **nonlinear activation functions** are required.

In present practice, four types of activation functions are generally used for fully connected networks.

1. **Linear** activation is used for the output layer of regression neural networks.
2. The **rectilinear** activation function is used for most hidden units. The rectilinear activation function is often referred to as **ReLU**.
3. A **leaky rectilinear** activation acts like a ReLU function for positive inputs, but has a small negative bias or leakage for negative input values. The leaky ReLU activation function can improve training for some deep neural networks.
4. The **logistic** or **sigmoid** activation function is used for binary classifiers.

5. The **softmax** activation function is used for multi-class classifiers.

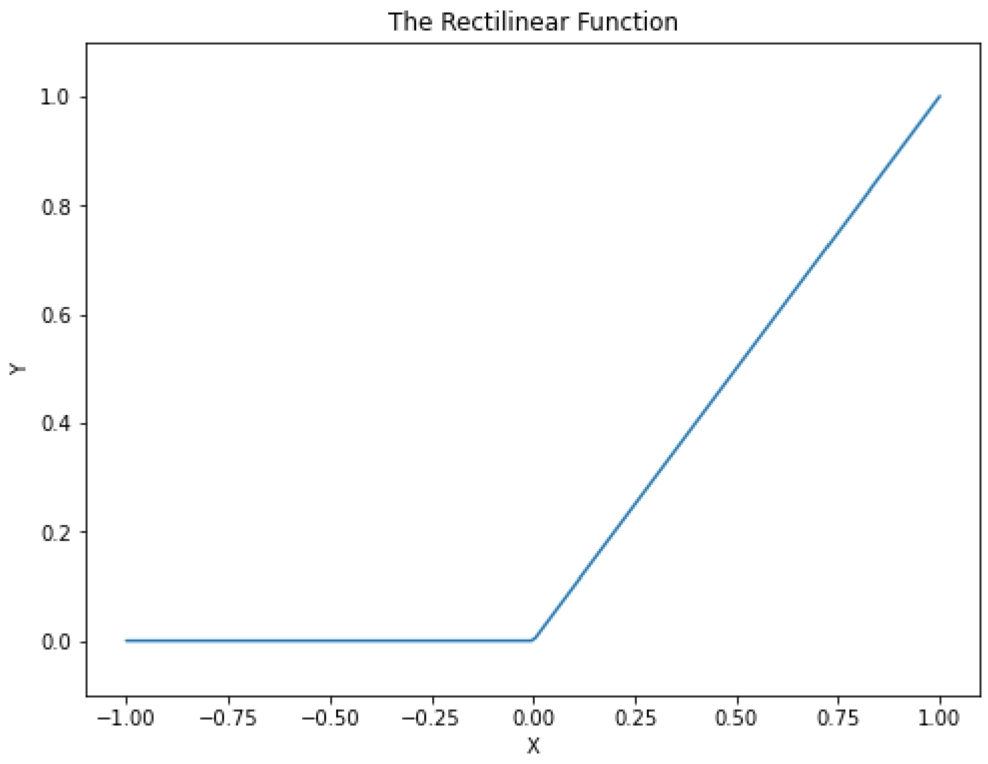
Rectilinear functions are typically used as the activation function for hidden units in neural networks. The rectilinear function is defined at:

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

The rectilinear function is linear for positive responses and zero for responses less than 0.0. Notice that the derivatives of the rectilinear function are not continuous. While this might seem to be a problem, in practice, even gradient-based optimization functions work well with this activation function.

The rectilinear function is plotted in the cell below:

In [ ]:
```
def reclu(x): return(max(0,x))

def plot_figs(x,y,title, figsize = (8, 6)):
    plt.figure(figsize=figsize).gca() # define axis
    sns.set_style("darkgrid")
    plt.plot(x, y)
    plt.ylim((-0.1,1.1))
    plt.title(title)
    plt.xlabel('X')
    plt.ylabel('Y')

x = np.linspace(-1.0, 1.0, 200)
y = [reclu(y) for y in x]
plot_figs(x,y,'The Rectilinear Function')
```



Another widely used activation function is the **logistic** or **sigmoid**. The sigmoid is used as the activation for the output layer of a binary classifier. The general sigmoid function can be written as:

$$\sigma(x) = \frac{L}{1 + e^{-k(x_0 - x)}}\\
where\\
L = max\ value\\
k = slope\\
x_0 = sigmoid\ midpoint$$

With $L = 1$, $k = 1$, and $x_0 = 0$, the logistic function becomes:

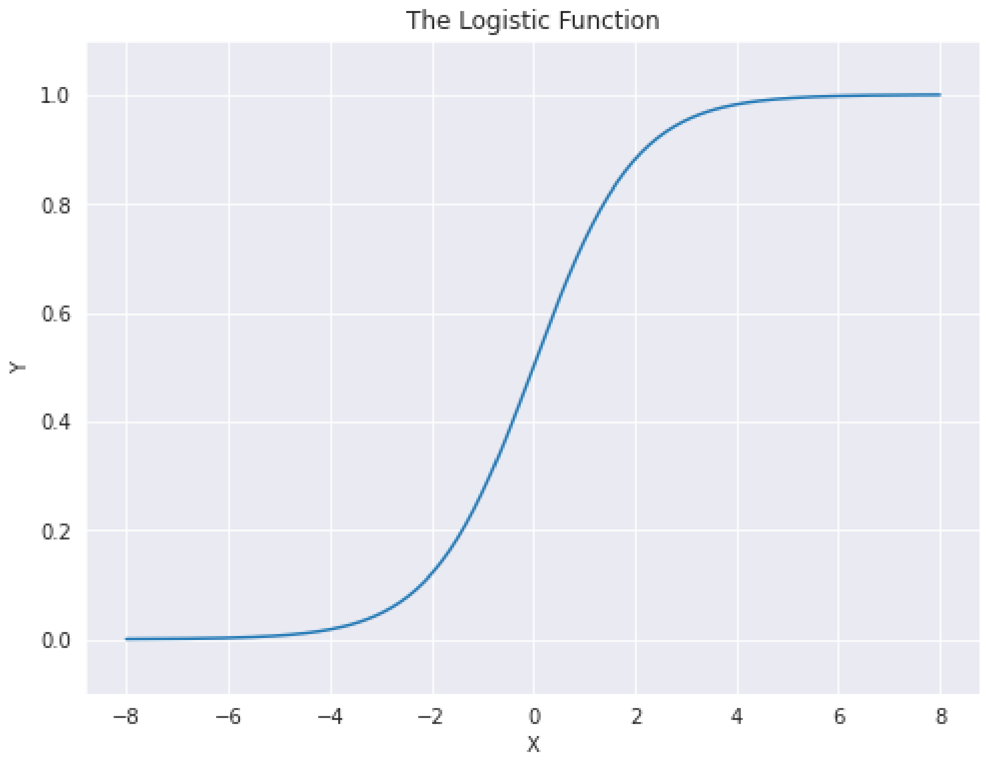
$$\sigma(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{1 + e^x}$$

The sigmoid function can asymptotically approach 0 or 1, but will never reach these extreme values. However, because of the rapid decrease in the derivative away from 0 the sigmoid can **saturate** when using gradient-based training. For this reason, the sigmoid is typically not used for hidden layers in neural networks.

When used in a the binary classifier a threshold is set to determine if the result is 0 or 1. The threshold can be adjusted to bias the result as desired.

The code in the cell below plots the sigmoid function.

```
In [ ]: def sigmoid(x): return exp(x)/(1 + exp(x))

        x = np.linspace(-8.0, 8.0, 200)
        y = [sigmoid(y) for y in x]
        plot_figs(x,y,'The Logistic Function') #, figsize = (5,3))
```



The **softmax** function or **normalized exponential function** is used for the output activation function of a multi-class classifier. The softmax function is the multinomial generalization of the sigmoid or logistic function. The probability of each class j is written as:

$$\sigma(z_j) = \frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}}$$

The normalization $\sum_{k=1}^K e^{z_k}$ ensures the sum of probabilities for all classes add to 1.0. The class selected by the classifier is the class with the largest value of $\sigma(z_j)$.

2.7 Computational example

Now that we have gone though some basic theory for feed-forward networks, it's time to try a simple example. You will construct a fully connected network to compute this simple function:

$$y = x_1 - x_2$$

Comment. You likely have noticed that this function is linear and can be computed easily without a neural network. Of course, that is not the point. We use a simple function to make the results easy to understand.

> **Exercise 5-1:** You will create and test a simple neural network implemented using matrix multiplication with Numpy. The architecture of the neural network is similar to the one shown in Figure 2.3, with 2 input units, 2 hidden units and 1 output unit. There are a total of 6 weights in two tensors. The neural network for this example does not require any bias terms.
>
> 1. As a first step, create test data for 3 possibilities; $x_1 > x_2$, $x_1 = x_2$, and $x_1 < x_2$, and with positive and negative values, or $x = [(2,1),(1,1),(1,2),(0,0),(2,-1),(-1,-1),(-2,1),(-1,-2)]$ as the input tuples.
> 2. Directly compute and print the evaluation of the function, $y = x_1 - x_2$, for each tuple.
>
> **Note:** The network you are asked to construct is simple and all weights must be in the set $\{-1, 1\}$. You can take advantage of the symmetry of the function. You must approximate to determine these weights by inspection. If you wish, you will find it easy to compute the partial derivatives of the function to be approximated. However, this is not necessary if you carefully inspect the network and consider the responses required.

In [ ]:
```
## Your code goes here
x = [(2,1), (1,1), (1,2), (0,0), (2,-1), (-1,-1), (-2,1), (-1,-2)]
```

```
for x_in in x:
    print(x_in[0] - x_in[1])
```

```
1
0
-1
0
3
0
-3
1
```

1. Now that you have the test data you can move to the next step. Determine the values of the 2×2 weight tensor between the input layer and the hidden layer (input tensor) and the 2×1 weight tensor between the hidden and output layer (output tensor). In the code cells below create as Numpy arrays and print these tensors. *Hint:* Keep in mind that the input tensor must be symmetric, with correct signs on the $\{-1, 1\}$ weights. Likewise, the output tensor $\{-1, 1\}$ weights must have the opposite signs.

In [ ]:
```
## Your code goes here
W_1 = np.array([[1.0, -1.0], [-1.0, 1.0]])
print(W_1)
```

```
[[ 1. -1.]
 [-1.  1.]]
```

In [ ]:
```
## Your code goes here
W_2 = np.array([1, -1])
print(W_2)
```

```
[ 1 -1]
```

1. Now, it is time to compute the results and check them. To create the computational process follow the graph in Figure 2.6, but ignoring the bias terms, b^1 and b^2. Create a function, `hidden`, to compute the output of the hidden layer using the formulation with **rectalinear activation**, $\delta()$:

$$h = \delta(W^1 \cdot x)$$

   Create a second function , `output`, to compute the vector product of the weight vector with the output vector of the hidden layer using a **linear activation**:

$$o = W^2 \cdot h$$

2. Execute the two functions while iterating over the input tuples and verify the output is correct. If not, reconsider the values of your weight tensors.

In [ ]:
```
## Your code goes here
def hidden(x, W):
    """Computes the output of the hidden layer"""
    h = np.dot(W, x) # product of weights and input vector
    return np.array([reclu(x) for x in h]) # apply activation function and retu

def output(h, W):
    """Computes the result for the hidden layer"""
    return np.dot(W, h) # dot product of weight vector and input vector

## Run the test cases and check the results
for y in x:
        h = hidden(y, W_1)
        print(output(h, W_2))
```

```
1.0
0
-1.0
0
3.0
0
-3.0
1.0
```

> If your results agree with the function created above, congratulations! Your first fully connected neural network passed all the tests!
> **End of exercise**.

Notice that even a network to compute a simple function requires 6 weights. You can see that for more complex functions any practical algorithm must learn a large number of weights. The limitations of Numpy would quickly become evident for large scale problems involving hundreds of millions of weights.

Note: If you are having difficulty following the Numpy code in the above example, you might want to look at Scott Shell's Numpy Tutorial

---

> **Exercise 5-2:** You will now construct and test a neural network implementing an exclusive or function, the XOR. The XOR function outputs a 1 if either input is 1 and the other 0, and a 0 otherwise. The truth table for the XOR function is:

$$\begin{bmatrix} in_1 & in_2 & out \\ 0 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix}$$

> You can use the `hidden` and `output` functions you created for the previous exercise with new weight tensors. Make sure you try all 4 possible test cases.

In [ ]:
```
x = [(0,0), (1,0), (0,1), (1,1)]

W_1 = np.array([[1.0, -1.0], [-1.0, 1.0]])

W_2 = np.array([1, 1])

## Run the test cases and check the results
for y in x:
        h = hidden(y, W_1)
        print(output(h, W_2))
```

```
0
1.0
1.0
0
```

> If your output agrees with the truth table, congratulations! You have solved the XOR problem using nonlinear activation.
> **End of exercise.**

3.0 Learning in neural networks: Backpropagation

Now that we have a promising representation, we need to determine if it is trainable. The answer is not only yes we can, but that we can do so in a computationally efficient manner, using a cleaver algorithm known as **backpropagation**.

The backpropagation algorithm was developed independently multiple times. The earliest work on this algorithm was by Kelly (1960) in the context of control theory and Bryson (1961) in the context of dynamic programming. Rumelhart, Hinton and Williams (1984) demonstrated empirically that backpropagation can be used to train neural networks. Their paper marks the beginning of the modern history of neural networks, and set off the first wave of enthusiasm.

The backpropagation algorithm requires several components. First, we need a **loss function** to measure how well our representation matches the function we are trying to learn. Second, we need a way to propagate changes in the representation through the complex network. For this we will use the **chain rule of calculus** to compute **gradients** of the representation. In the general case, this process requires using automatic differentiation methods.

The point of backpropagration is to learn the optimal weight for the neural network. The algorithm proceeds iteratively through a series of small steps. Once we have the gradient of the loss function we can update the tensor of weights.

$$W_{t+1} = W_t + \alpha \nabla_W J(W_t)$$

where
$W_t =$ the tensor of weights or model parameters at step t.
$\alpha \ =$ step size or learning rate.
$J(W) =$ loss function given the weights.
$\nabla_W J(W) =$ gradient of J with respect to the weights W.

It should be evident that the back propagation algorithm is a form of gradient decent. The weights are updated in small steps following the gradient of $J(W)$ down hill.

Finally, we need a way evaluate the performance of the model. Without evaluation metrics we have no way to compare the performance of a given model, or compare the performance of several models.

In the next sections, we will address each of loss functions, gradient computation and performance measurement.

3.1 Loss functions

To train a neural network we must have a **loss function**, also known as a **cost function**. In simple terms, the loss function measures the fit of a model to the training data. The lower the loss, the better the fit.

To train deep learning models **cross entropy** is often used as a loss function. This is an information theoretic measure of model fit. We can understand cross entropy as follows.

First define **Shannon entropy** as:

$$\mathbb{H}(I) = E[I(X)] = E[-ln_b(P(X))] = - \sum_{i=1}^n P(x_i) ln_b(P(x_i)$$

Where:
$E[X] =$ the expectation of X.
$I(X) =$ the information content of X.
$P(X) =$ probability of X.
$b =$ base of the logarithm.

This rather abstract formula gives us a way to compute the expected information content of a set of values X. The more likely (higher probability) of X the less informative it is.

To create a loss function from the definition of Shannon entropy we start with the **Kullback-Leibler divergence (KL divergence)** or **relative entropy**. The KL divergence is an information theoretic measure of the difference between two distributions, $P(X)$ and $Q(X)$
.

$$\mathbb{D}_{KL}(P \parallel Q) = -\sum_{i=1}^{n} p(x_i)\ ln_b \frac{p(x_i)}{q(x_i)}$$

Ideally, in the case of training a machine learning model we want a distribution $Q(X)$, which is identical to the actual data distribution $P(X)$.

But, you may say, if we could know $P(X)$ why compute $Q(X)$ at all? Fortunately, we do not have to. We can rewrite the KL divergence as:

$$\mathbb{D}_{KL}(P \parallel Q) = \sum_{i=1}^{n} p(x_i)\ ln_b p(x_i) - \sum_{i=1}^{n} p(x_i)\ ln_b q(x_i)$$

Since $P(X)$ is fixed and we wish to find $Q(X)$ when we train our model, we can minimize the term on the right, which is the **cross entropy** defined as:

$$\mathbb{H}(P,Q) = -\sum_{i=1}^{n} p(x_i)\ ln_b q(x_i)$$

From the formulation of KL divergence above you can see the following.

$$\mathbb{D}_{KL}(P \parallel Q) = \mathbb{H}(P) + \mathbb{H}(P,Q)$$
$$\mathbb{D}_{KL}(P \parallel Q) = Entropy(P) + Cross\ Entropy(P,Q)$$

Thus, we can minimize divergence by minimizing cross entropy. This idea is both intuitive and computationally attractive. The closer the estimated distribution $q(X)$ is to the distribution of the true underling process $p(X)$, the lower the cross entropy and the lower the KL divergence.

In general we will not know $p(X)$. In fact, if we did, why would we need to solve a training problem? So, we can use the following approximation.

$$\mathbb{H}(P,Q) = -\frac{1}{N}\sum_{i=1}^{n} ln_b q(x_i)$$

You may notice, that this approximation, using the average log likelihood, is equivalent to a maximum likelihood estimator (MLE).

Let's look at a specific case of a model with Gaussian likelihood. What is the cross entropy? We can start by thinking about the definition of likelihood.

$$p(data|model) = p(data|f(\theta)) = p(x_i|f(\hat{\mu},\sigma)) = \frac{1}{2\pi\sigma^2} e^{\frac{-(x_i-\hat{\mu})^2}{2\sigma^2}}$$

We take the negative logarithm of this likelihood model.

$$-log\big(p(data|model)\big) = -\frac{1}{2}\big(log(2\pi\sigma^2) + \frac{(x_i-\hat{\mu})^2}{2\sigma^2}\big)$$

Now, the first term on the right is a constant, as is the denominator of the second term if we assume known variance. Since our goal is to minimize cross entropy, we can eliminate these quantities and be left with just the following.

$$-(x_i - \hat{\mu})^2$$

This is one issue we need to deal with. Our formulation of cross entropy involves the unknown true distribution of the underling process $p(X)$. However, since $p(x_i)$ is fixed but unknown we can just write the following.

$$min\big(\mathbb{H}(P,Q)\big) \propto argmin_{\mu}\big(-\sum_{i=1}^n (x_i - \hat{\mu})^2\big)$$

This is just the definition of a Maximum Likelihood Estimator (MLE) for the least squares problem! In fact, since the cross entropy is computed using the negative log likelihood, it will always be minimized by the MLE.

You can see another example of cross-entropy error function and logistic regression.

3.2 Computing Loss functions

The loss function is used to train the model. Therefore the loss function must be computed in an efficient manner.

Given the number of parameters in deep neural nets over-fitting is inevitable. Therefore some regularization is required. We will discuss regularization in greater depth in another lesson. For now, we will just use the following regularized form.

$$\mathbb{H}(P,Q) = J(\theta) = -\frac{1}{N}\sum_{i=1}^n ln_b q(x_i|\theta) + \lambda||\theta||^2$$
$$where$$
$$-\frac{1}{N}\sum_{i=1}^n ln_b q(x_i|\theta) = J_{MLE}(\theta)$$
$$||\theta||^2 = L^2\ norm\ regularization\ term$$

To minimize $J(\theta)$ in this form θ must be chosen to keep $||\theta||^2$ small while minimizing the negative log likelihood of $q(x_i|\theta)$.

Let's consider how we would compute this form of the lost function. The computational graph shown below illustrates the computational path for the regularize d loss function. For simplicity, no bias terms are considered.

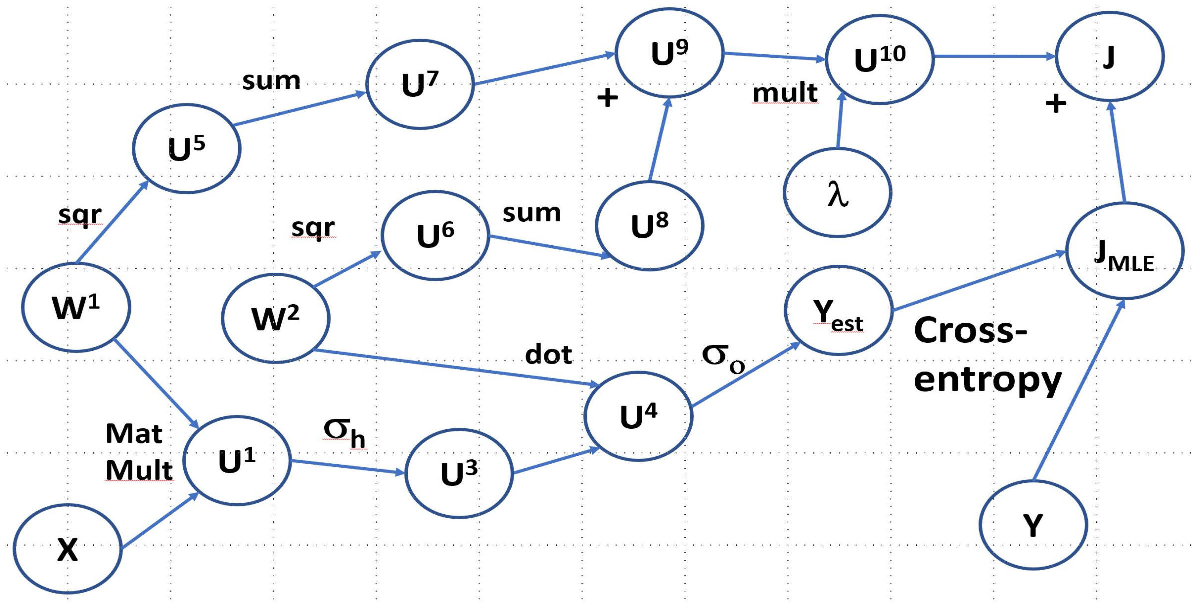


Figure 3.1 Computational graph for computing loss of fully connected neural network of Figure 2.3

```
In [ ]: from google.colab import drive
        drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

3.3 Chain rule of calculus

Key to the back propagation algorithm is the chain rule of calculus; not to be confused with the chain rule of probability. The chain rule allows us to back propagate gradients though an arbitrarily complex graph of functions.

Now, suppose there is a function $y = g(x)$, and another function $z = f(y) = f(g(x))$. How do we compute the derivative of z with respect to x? Applying the chain rule we get:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

Consider $x \in R^M$ $g(x) \Rightarrow R^M$ and $f(y) \Rightarrow z \in R$. The chain rule becomes:

$$\frac{\partial z}{\partial x} = \sum_{j \in M} \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}$$

Which we can rewrite as

$$\nabla_x z = \left(\frac{\partial x}{\partial y}\right)^T \nabla_y z$$

Here, $\frac{\partial x}{\partial y}$ is the nxm **Jacobian matrix** of partial derivatives. The Jacobian is multiplied by the gradient with respect to y, $\nabla_y z$. You can think of the Jacobian as a transformation for a gradient with respect to y to what we really want, the gradient with respect to z.

3.4 Example of finding a gradient.

Let's work out backpropagation for a very simple neural network with a just an input layer and an output layer. This neural network, including the loss function, is shown in Figure 3.2 below. This network has been highly simplified. There are only three layers, input layer, a two unit hidden layer with no bias terms, and a single unit output layer. There are only two weight tensors for this network. Further, the hidden units use rectilinear activation and the output unit uses linear activation. These activation functions have simple partial derivatives.

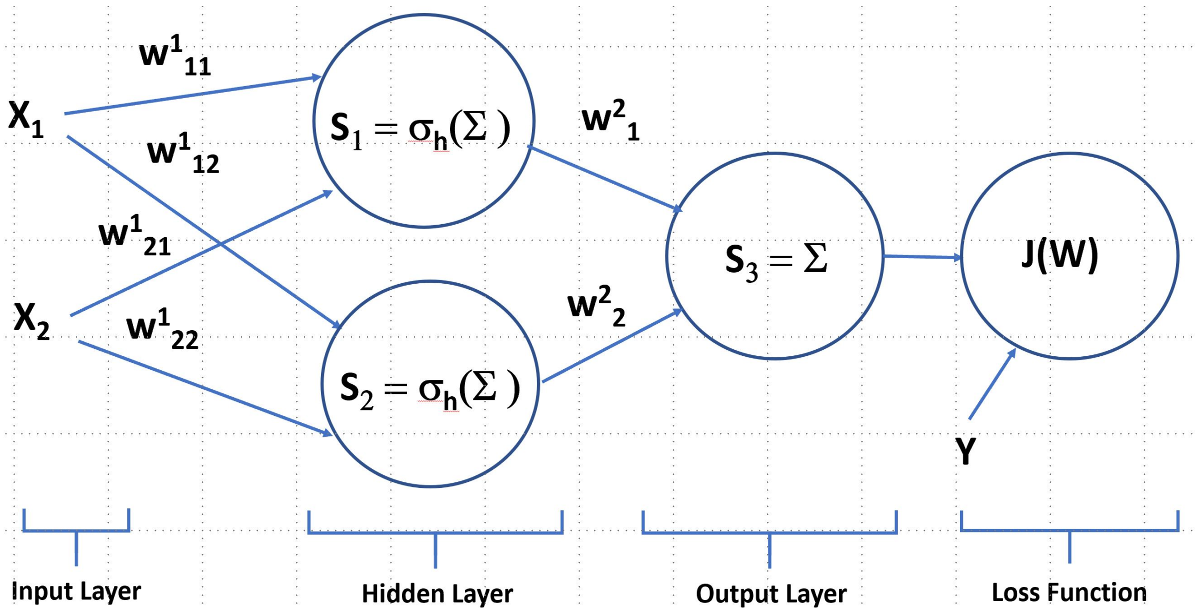


Figure 3.2 Simple single layer neural network with loss function

To analyze this network we will refer to the computational graph shown in Figure 3.1 above.

First, we need to work out the forward propagation relationships. We can compute the outputs of the hidden layer as follows.

$$S_{\{1,2\}} = \sigma_h\big(W^1 \cdot X_{\{1,2\}}\big) = \sigma\Big(\sum_j W^1_{i,j} x_j\Big)$$

In the same way, the result from the output layer can be computed as follows, since the activation function for this layer is linear.

$$S_3 = W^2 \cdot S_{\{1,2\}} = \sum_i W^2_i \sigma\Big(\sum_j W^1_{i,j} x_j\Big)$$

To perform backpropagation, we need fill out the gradient vector by computing $\frac{\partial J(W)}{\partial W}$ for each weight in the model.

$$\frac{\partial J(W)}{\partial W} = \begin{bmatrix} \frac{\partial J(W)}{\partial W^2_{11}} \\ \frac{\partial J(W)}{\partial W^2_{12}} \\ \frac{\partial J(W)}{\partial W^2_{21}} \\ \frac{\partial J(W)}{\partial W^2_{22}} \\ \frac{\partial J(W)}{\partial W^1_{1}} \\ \frac{\partial J(W)}{\partial W^1_{2}} \end{bmatrix}$$

To keep things simple in this example we will just use a non-normalized squared error loss function. This is just the MLE estimator (without normalization) for a Gaussian distribution.

$$J(W) = -\frac{1}{2} \sum_{l=1}^{n} (y_l - S_{3,l})^2$$

Where:
$y_k =$ the label for the lth case.
$\hat{y_k} = S_{3,k} =$ the output of the network for the lth case.

We want to compute the gradients with respect to the input and output tensors:

$$\frac{\partial J(W)}{\partial W^1}, \frac{\partial J(W)}{\partial W^2}$$

Let's start with the easier case of the partial derivatives with respect to the output tensor. We can apply the chain rule as follows:

$$\frac{\partial J(W)}{\partial W^2_k} = \frac{\partial J(W)}{\partial S_{3,k}} \frac{\partial S_{3,k}}{\partial W^2_k}$$

The first partial derivative of the chain is:

$$\frac{\partial J(W)}{\partial S_{3,k}} = \frac{\partial - \frac{1}{2}(y_k - S_{3,k})^2}{\partial S_{3,k}} = y_k - S_{3,k}$$

And, the partial derivative of the second partial derivative in the chain, given the linear activation of the output unit:

$$\frac{\partial S_{3,k}}{\partial W^2_k} = \frac{\partial W^2_k S_{j,k}}{\partial W^2_k} = S_{j,l},\ j \in \{1, 2\}$$

Multiplying the two components of the chain gives us:

$$\frac{\partial J(W)}{\partial W^2_k} = S_{j,k}(y_k - S_{3,k}),\ j \in \{1,2\}$$

The partial derivatives with respect to the input tensor are a bit more complicated. To apply the chain rule we must work backwards from the loss function. This gives the following chain:

$$\frac{\partial J(W)}{\partial W^1_{i,j}} = \frac{\partial J(W)}{\partial S_3} \frac{\partial S_3}{\partial S_j} \frac{\partial S_j}{\partial W^1_{i,j}}$$

First, we find the right most partial derivative in our chain:

$$\frac{\partial S_j}{\partial W^1_{i,j}} = \begin{cases} \frac{\partial W^1_{i,j} x_{i,k}}{\partial W^1_{i,j}}, & \text{if } S_j > 0 \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

Which given the ReLU activation results in:

$$\frac{\partial S_j}{\partial W^1_{i,j}} = \begin{cases} 1, & \text{if } S_j > 0 \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

The middle partial derivative must account for the nonlinearity:

$$\frac{\partial S_3}{\partial S_j} = W^2_j$$

We have already computed $\frac{\partial J(W)}{\partial S_3}$. Multiplying all three partial derivatives we find:

$$\frac{\partial J(W)}{\partial W^1_{i,j}} = \begin{cases} (y_k - S_{3,k}) W^2_j, & \text{if } S_j > 0 \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

Where S_3 and $S_{\{1,2\}}$ are computed using the relationships given above.

A more detailed, but still digestable example of computing gradients for backpropagation can be found in a blog post by Manfred Zaharauskas, among many other places.

4.0 Creating a Model With Keras

You will now create and test a first deep learning classifier model for the MNIST dataset using Keras. The fully connected model has one hidden layer.

4.1 Preparing the Dataset

You have already worked with the MNIST dataset. To load these data and prepare them for the Keras model execute the code in the cell below.

In [ ]:
```
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
train_images = train_images.reshape((60000, 28*28)).astype('float32')/255
print(train_images.shape)
test_images = test_images.reshape((10000, 28*28)).astype('float32')/255
print(test_images.shape)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datas
ets/mnist.npz
11490434/11490434 [==============================] - 0s 0us/step
(60000, 784)
(10000, 784)
```

In [ ]:
```
train_images.dtype
```

Out[ ]:
```
dtype('float32')
```

There is one more preprocessing step. The labels need to be **one hot encoded**. One hot encoding transforms an N level categorical variable into N binary columns. One column represents one category. A 1 or binary true value is encoded in the column of a given category with the other columns coded as 0 or false.

> **Exercise 5-3:** You will now one hot encoded the label vectors of the training and test data. Use the keras.utils.np_utils.to_categorical function to create the one hot encoded labels. Print the first 10 rows of the training labels. You will need to set options to display all columns.

In [ ]:
```
## Put your code here
train_labels = tf.keras.utils.to_categorical(train_labels)
test_labels = tf.keras.utils.to_categorical(test_labels)

print("First 10 rows:\n")
with np.printoptions(threshold=np.inf):
    print(train_labels[:10,:])
```

```
First 10 rows:

[[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]]
```

> Examine the printed one hot encoded labels. Does the number of columns correspond to the number of label categories and why is this expected?
> **End of exercise.**

> **Answer:**

> Yes, it does. It is expected because the dataset has 10 different digits. Here, each column represents a digit due to one-hot encoding.

4.2 Defining and Executing the Deep Learning Model

It is now time to define the Keras neural network model. This model uses the Keras **Sequential** class. The model is constructed by **adding dense layers** with the `add` method. Hidden and output layers are specified by creating instances of the Dense layer class. An input data shape must be specified only for the input layer.

> **Exercise 5-4:** You will now specify, compile and fit the neural network model by the following steps:
>
> 1. Define the sequential model by instantiating a model object using `models.Sequential`. Name your model `nn`. Then add layers:
>   - Add a dense input layer with 28×28 hidden units, rectalinear (`relu`) activation and `input_shape=(28*28, )`.
>   - Add a dense hidden layer with 512 hidden units and rectalinear (`relu`) activation.
>   - Add a dense output output layer with `activation='softmax'`.
> 2. Compile your model with the following arguments; `optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy']`. We will discuss optimizers in another lesson.
> 3. Print a summary of the model with the `summary()` method.
> 4. Fit your model, using the fit() method, with the training images, training labels, and arguments; `epochs=5, batch_size=128`.

In [ ]:
```
## Your code goes here

# 1. Define the sequential model by instantiating a model object using models.S
#    - Add a dense input layer with  28×28  hidden units, rectalinear (relu) ac
#    - Add a dense hidden layer with 512 hidden units and rectalinear (relu) ac
#    - Add a dense output output layer with activation='softmax'.

nn = models.Sequential()
nn.add(layers.Dense(28 * 28, activation = "relu", input_shape = (28 * 28, )))
nn.add(layers.Dense(512, activation = "relu"))
nn.add(layers.Dense(len(train_labels[0]), activation = "softmax"))


# 2. Compile your model with the following arguments; optimizer='rmsprop',
#    loss='categorical_crossentropy', metrics=['accuracy']. We will discuss
#    optimizers in another lesson.

nn.compile(optimizer = "rmsprop", loss = 'categorical_crossentropy', metrics =


# 3. Print a summary of the model with the summary() method.
```

```
nn.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 784)               615440

 dense_1 (Dense)             (None, 512)               401920

 dense_2 (Dense)             (None, 10)                5130

=================================================================
Total params: 1,022,490
Trainable params: 1,022,490
Non-trainable params: 0
_________________________________________________________________
```

In [ ]:

```
## Your code goes here

# 4. Fit your model, using the fit() method, with the training images, training
#    labels, and arguments; epochs=5, batch_size=128.

nn.fit(train_images, train_labels, batch_size = 128, epochs = 5)
```

```
Epoch 1/5
469/469 [==============================] - 16s 31ms/step - loss: 0.2205 - accu
racy: 0.9318
Epoch 2/5
469/469 [==============================] - 14s 31ms/step - loss: 0.0788 - accu
racy: 0.9752
Epoch 3/5
469/469 [==============================] - 14s 31ms/step - loss: 0.0498 - accu
racy: 0.9840
Epoch 4/5
469/469 [==============================] - 13s 28ms/step - loss: 0.0341 - accu
racy: 0.9890
Epoch 5/5
469/469 [==============================] - 13s 28ms/step - loss: 0.0256 - accu
racy: 0.9921
```

Out[ ]:

```
<keras.callbacks.History at 0x7fbba53c3160>
```

> Answer these questions:
>
> 1. Why is the `categorical_crossentropy` the good choice loss function and `softmax` the correct choice for output activation for this model?
> 2. Considering the number of training samples, and number of trainable model parameters, where do you think this model might lie along the bias-variance trade-off spectrum?
> 3. Examine the evolution of the loss function and accuracy. What do these figures tell you about learning for this model?
>
>   **End or exercise.**

Answers:

1. The output layer of our model is a softmax because there are 10 possible values for the answer. softmax returns the probability distribution for these possible outputs, where the sum of the ten scores is 1. The prediction to be selected is the class with the highest probability. categorical_crossentropy is the best choice because it measures the distance between probability distribution output and the true distribution of the labels.

2. We have to consider that there are 60,000 images and 1,022,490 parameters. Given that the number of parameters is several orders of magnitude larger than the samples, the model may be prone to overfit.

3. The accuracy and loss values returned indicate that the model learned very well the training data. That is, the model will correctly predict a value taken from the training data 99.21% of the time. However, we still have to verify how the model predicts unkown test data. If accuracy for test data predictions is low, that means that the model is overfitting.

4.3 Performance Metrics

Now that we have the components for training a basic neural network in place we need a way to evaluate its performance. It turns out, there is nothing special about evaluation of neural network models as opposed to other machine learning models. For regression models, one typically use the standard metrics such as root mean square error (RMSE), mean absolute error (MAE). For classification models, one also typically uses the standard metrics including the confusion matrix, accuracy, precision and recall. The Keras metrics package provides numerous methods for model evaluation.

Execute the code in the cell below to compute and display performance metrics for your model based on the test dataset.

In [ ]:
```
test_loss, test_accuracy = nn.evaluate(test_images, test_labels)
print("Test accuracy = {0:.3f}    Test loss = {1:.3f}".format(test_accuracy, tes
```

```
313/313 [==============================] - 3s 8ms/step - loss: 0.0797 - accura
cy: 0.9769
Test accuracy = 0.977   Test loss = 0.080
```

Exercise 5-5: Compare the results of the evaluation with the same metrics achieved during model training. What does the difference tell you about the generalization for this model?

Answer:

> The difference tells me how well the model generalizes. It is normal that accuracy is higher and loss is lower for training data because training samples are used to train the model. In this case, the accuracy of 98.14% and the loss of 0.0633 tells me that the model is doing wonderful. It is able to generalize very well to correctly predict unknowm samples without overfitting issues.

5.0 Using Training History

The Keras model `fit` method creates a TensorFlow history object using callbacks. The information contained in the history object can be very useful in understanding model training; what is working and what is not.

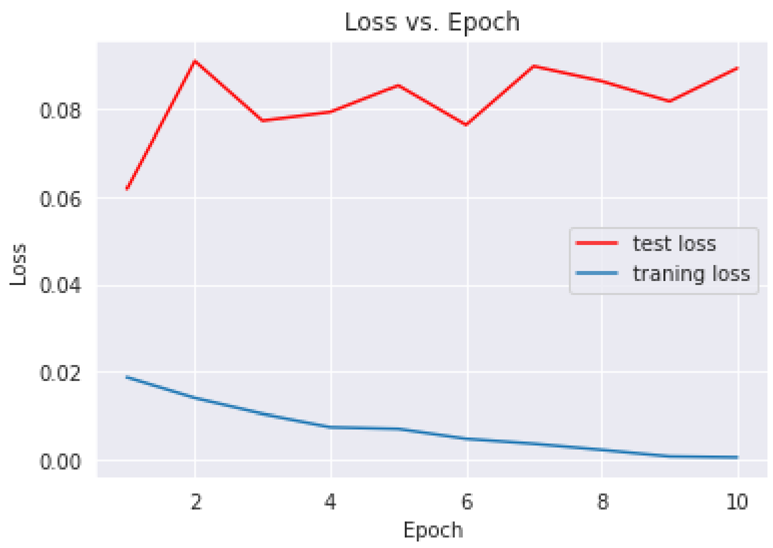
> **Exercise 5-6:** You will now re-train your model while capturing a history. To retrain a model you must re-compile it first to create a fresh model object. Training an existing model object will continue the training of that object. This property of Keras models can be most useful for improving existing models when new training data becomes available. Do the following:
>
> 1. Compile the existing model using the same arguments as before.
> 2. Fit the model as before, but for 10 epochs and with an additional argument; `validation_data=(test_images, test_labels)`. Assign the results to a variable, `history_nn`.
> 3. Execute the code in the next two cells to display charts of training and test loss and accuracy.

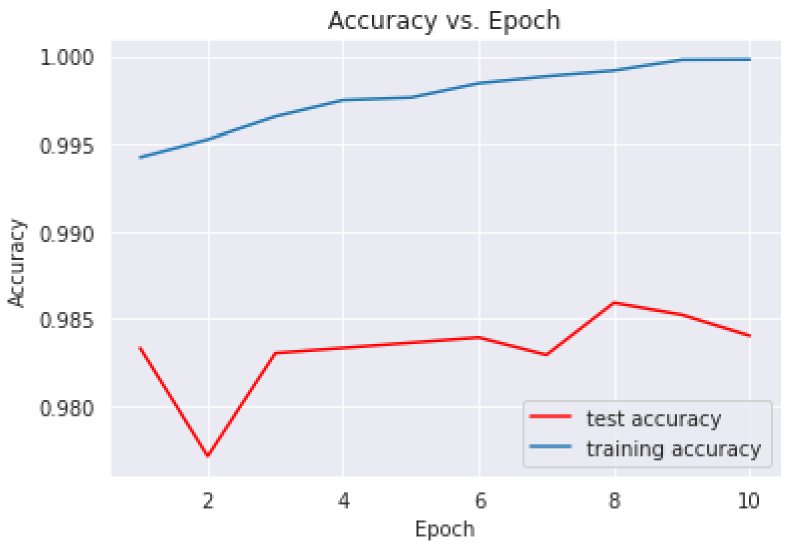
In [ ]:
```
## Your code goes here

# 1. Compile the existing model using the same arguments as before.

nn.compile(optimizer = "rmsprop", loss = 'categorical_crossentropy', metrics =


# 2. Fit the model as before, but for 10 epochs and with an additional argument
#    validation_data=(test_images, test_labels). Assign the results to a
#    variable, history_nn.

history_nn = nn.fit(train_images, train_labels, batch_size = 128, epochs = 10,


# 3. Execute the code in the next two cells to display charts of training and
#    test loss and accuracy.
```

```
Epoch 1/10
469/469 [==============================] - 16s 31ms/step - loss: 0.0187 - accu
racy: 0.9942 - val_loss: 0.0618 - val_accuracy: 0.9833
Epoch 2/10
469/469 [==============================] - 16s 35ms/step - loss: 0.0140 - accu
racy: 0.9952 - val_loss: 0.0908 - val_accuracy: 0.9771
Epoch 3/10
469/469 [==============================] - 14s 31ms/step - loss: 0.0105 - accu
racy: 0.9966 - val_loss: 0.0772 - val_accuracy: 0.9830
Epoch 4/10
469/469 [==============================] - 15s 32ms/step - loss: 0.0074 - accu
racy: 0.9975 - val_loss: 0.0792 - val_accuracy: 0.9833
Epoch 5/10
469/469 [==============================] - 15s 32ms/step - loss: 0.0071 - accu
racy: 0.9976 - val_loss: 0.0852 - val_accuracy: 0.9836
Epoch 6/10
469/469 [==============================] - 16s 34ms/step - loss: 0.0048 - accu
racy: 0.9984 - val_loss: 0.0762 - val_accuracy: 0.9839
Epoch 7/10
469/469 [==============================] - 15s 33ms/step - loss: 0.0037 - accu
racy: 0.9988 - val_loss: 0.0897 - val_accuracy: 0.9829
Epoch 8/10
469/469 [==============================] - 14s 30ms/step - loss: 0.0023 - accu
racy: 0.9992 - val_loss: 0.0862 - val_accuracy: 0.9859
Epoch 9/10
469/469 [==============================] - 15s 32ms/step - loss: 7.9281e-04 -
accuracy: 0.9998 - val_loss: 0.0816 - val_accuracy: 0.9852
Epoch 10/10
469/469 [==============================] - 15s 32ms/step - loss: 5.8799e-04 -
accuracy: 0.9998 - val_loss: 0.0892 - val_accuracy: 0.9840
```

In [ ]:

```python
def plot_loss(history):
    train_loss = history.history['loss']
    test_loss = history.history['val_loss']
    x = list(range(1, len(test_loss) + 1))
    plt.plot(x, test_loss, color = 'red', label = 'test loss')
    plt.plot(x, train_loss, label = 'traning loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.title('Loss vs. Epoch')
    plt.legend()
    plt.show()

def plot_accuracy(history):
    train_acc = history.history['accuracy']
    test_acc = history.history['val_accuracy']
    x = list(range(1, len(test_acc) + 1))
    plt.plot(x, test_acc, color = 'red', label = 'test accuracy')
    plt.plot(x, train_acc, label = 'training accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.title('Accuracy vs. Epoch')
    plt.legend(loc='lower right')
    plt.show()

plot_loss(history_nn)
plot_accuracy(history_nn)
```





> Examine the trajectory of the training and test loss and accuracy. What does the divergence of these losses and accuracies tell you about the learning and generalization of the model?
> **End of exercise.**

> **Answer:**
>
> Although the range within accuracy oscilates is very small and for loss is not that different, the oscilating behavior of loss and accuracy for test data while loss steadily decreases and accuracy steadily increases for train data is a sign of overfitting.

6.0 Adding Regularization

You have seen some of the effects of over-fitting of the neural network model. **Regularization** methods are widely used in machine learning to prevent over-fitting.

Conceptually, you can think of regularization as moving the model toward lower variance and higher bias to improve generalization. We will examine regularization in greater detail in another lesson.

Keras used the layer weight regularizer class to add weight constraints to the layers. Here, we will only used the L2 regularizer.

> **Exercise 5-7:** You will now add L2 regularization to the dense input and hidden layers of your model. Do the following:
>
> 1. Starting with the model specification you have been using, add the following argument to the input and hidden layer; `kernel_regularizer=regularizers.l2(0.1)` .
> 2. Compile the model.
> 3. Fit the model for 80 epochs, saving the history object. This will take some time!
> 4. Plot the training and test loss and accuracy.

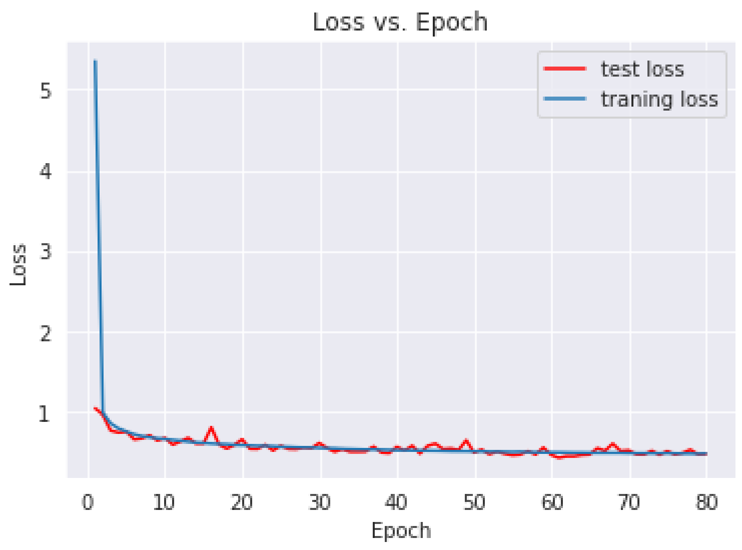
In [ ]:
```
## Your code goes here

# 1. Starting with the model specification you have been using, add the followi
#    argument to the input and hidden layer;
#    kernel_regularizer=regularizers.l2(0.1).

nn = models.Sequential()
nn.add(layers.Dense(28 * 28, activation = "relu", input_shape = (28 * 28, ), ke
nn.add(layers.Dense(512, activation = "relu", kernel_regularizer = regularizers
nn.add(layers.Dense(len(train_labels[0]), activation = "softmax"))


# 2. Compile the model.

nn.compile(optimizer = "rmsprop", loss = 'categorical_crossentropy', metrics =


# 3. Fit the model for 80 epochs, saving the history object. This will take son

history_nn = nn.fit(train_images, train_labels, batch_size = 128, epochs = 80,


# 4. Plot the training and test loss and accuracy.

plot_loss(history_nn)
plot_accuracy(history_nn)
```

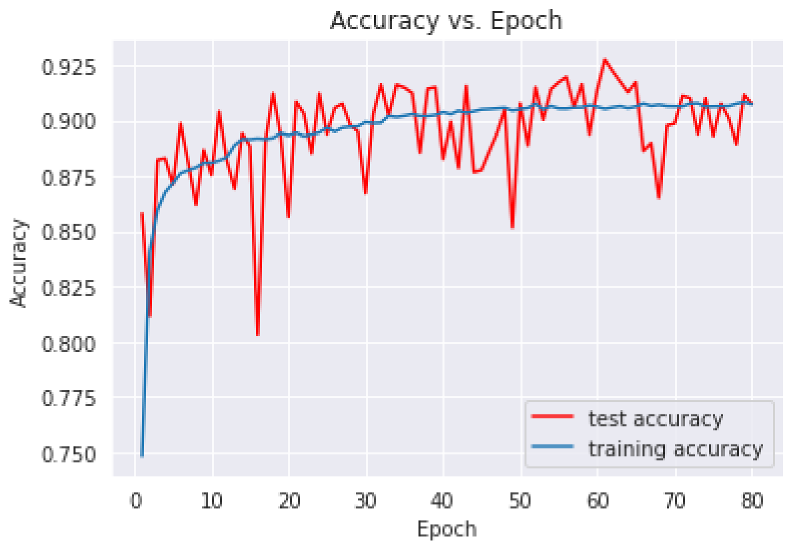
```
Epoch 1/80
469/469 [==============================] - 18s 36ms/step - loss: 5.3530 - accu
racy: 0.7480 - val_loss: 1.0355 - val_accuracy: 0.8581
Epoch 2/80
469/469 [==============================] - 16s 34ms/step - loss: 0.9860 - accu
racy: 0.8408 - val_loss: 0.9584 - val_accuracy: 0.8115
Epoch 3/80
469/469 [==============================] - 17s 37ms/step - loss: 0.8553 - accu
racy: 0.8596 - val_loss: 0.7647 - val_accuracy: 0.8823
Epoch 4/80
469/469 [==============================] - 15s 33ms/step - loss: 0.7923 - accu
racy: 0.8677 - val_loss: 0.7398 - val_accuracy: 0.8830
Epoch 5/80
469/469 [==============================] - 15s 33ms/step - loss: 0.7512 - accu
racy: 0.8717 - val_loss: 0.7502 - val_accuracy: 0.8712
Epoch 6/80
469/469 [==============================] - 17s 36ms/step - loss: 0.7182 - accu
racy: 0.8763 - val_loss: 0.6543 - val_accuracy: 0.8986
Epoch 7/80
469/469 [==============================] - 16s 34ms/step - loss: 0.6957 - accu
racy: 0.8777 - val_loss: 0.6673 - val_accuracy: 0.8808
Epoch 8/80
469/469 [==============================] - 16s 34ms/step - loss: 0.6811 - accu
racy: 0.8789 - val_loss: 0.7054 - val_accuracy: 0.8620
Epoch 9/80
469/469 [==============================] - 17s 36ms/step - loss: 0.6675 - accu
racy: 0.8809 - val_loss: 0.6410 - val_accuracy: 0.8868
Epoch 10/80
469/469 [==============================] - 16s 35ms/step - loss: 0.6584 - accu
racy: 0.8809 - val_loss: 0.6721 - val_accuracy: 0.8755
Epoch 11/80
469/469 [==============================] - 16s 35ms/step - loss: 0.6480 - accu
racy: 0.8820 - val_loss: 0.5890 - val_accuracy: 0.9040
Epoch 12/80
469/469 [==============================] - 16s 34ms/step - loss: 0.6380 - accu
racy: 0.8835 - val_loss: 0.6288 - val_accuracy: 0.8821
Epoch 13/80
469/469 [==============================] - 16s 33ms/step - loss: 0.6250 - accu
racy: 0.8889 - val_loss: 0.6734 - val_accuracy: 0.8693
Epoch 14/80
469/469 [==============================] - 17s 37ms/step - loss: 0.6161 - accu
racy: 0.8916 - val_loss: 0.6028 - val_accuracy: 0.8941
Epoch 15/80
469/469 [==============================] - 16s 34ms/step - loss: 0.6108 - accu
racy: 0.8915 - val_loss: 0.6042 - val_accuracy: 0.8885
Epoch 16/80
469/469 [==============================] - 15s 32ms/step - loss: 0.6030 - accu
racy: 0.8919 - val_loss: 0.8045 - val_accuracy: 0.8030
Epoch 17/80
469/469 [==============================] - 15s 33ms/step - loss: 0.5977 - accu
racy: 0.8916 - val_loss: 0.5895 - val_accuracy: 0.8906
Epoch 18/80
469/469 [==============================] - 16s 33ms/step - loss: 0.5944 - accu
racy: 0.8921 - val_loss: 0.5401 - val_accuracy: 0.9122
Epoch 19/80
469/469 [==============================] - 16s 35ms/step - loss: 0.5862 - accu
racy: 0.8943 - val_loss: 0.5850 - val_accuracy: 0.8929
Epoch 20/80
469/469 [==============================] - 16s 35ms/step - loss: 0.5833 - accu
racy: 0.8932 - val_loss: 0.6552 - val_accuracy: 0.8565
```

```
Epoch 21/80
469/469 [==============================] - 16s 34ms/step - loss: 0.5784 - accu
racy: 0.8945 - val_loss: 0.5378 - val_accuracy: 0.9084
Epoch 22/80
469/469 [==============================] - 16s 34ms/step - loss: 0.5740 - accu
racy: 0.8929 - val_loss: 0.5361 - val_accuracy: 0.9031
Epoch 23/80
469/469 [==============================] - 16s 34ms/step - loss: 0.5740 - accu
racy: 0.8937 - val_loss: 0.5874 - val_accuracy: 0.8853
Epoch 24/80
469/469 [==============================] - 16s 35ms/step - loss: 0.5691 - accu
racy: 0.8946 - val_loss: 0.5161 - val_accuracy: 0.9122
Epoch 25/80
469/469 [==============================] - 16s 33ms/step - loss: 0.5625 - accu
racy: 0.8968 - val_loss: 0.5741 - val_accuracy: 0.8938
Epoch 26/80
469/469 [==============================] - 15s 33ms/step - loss: 0.5609 - accu
racy: 0.8949 - val_loss: 0.5360 - val_accuracy: 0.9056
Epoch 27/80
469/469 [==============================] - 15s 33ms/step - loss: 0.5565 - accu
racy: 0.8967 - val_loss: 0.5337 - val_accuracy: 0.9075
Epoch 28/80
469/469 [==============================] - 15s 31ms/step - loss: 0.5532 - accu
racy: 0.8971 - val_loss: 0.5522 - val_accuracy: 0.8980
Epoch 29/80
469/469 [==============================] - 15s 32ms/step - loss: 0.5523 - accu
racy: 0.8973 - val_loss: 0.5440 - val_accuracy: 0.8951
Epoch 30/80
469/469 [==============================] - 16s 33ms/step - loss: 0.5469 - accu
racy: 0.8992 - val_loss: 0.6090 - val_accuracy: 0.8673
Epoch 31/80
469/469 [==============================] - 16s 34ms/step - loss: 0.5444 - accu
racy: 0.8987 - val_loss: 0.5470 - val_accuracy: 0.9027
Epoch 32/80
469/469 [==============================] - 17s 35ms/step - loss: 0.5427 - accu
racy: 0.8989 - val_loss: 0.5007 - val_accuracy: 0.9162
Epoch 33/80
469/469 [==============================] - 16s 34ms/step - loss: 0.5368 - accu
racy: 0.9021 - val_loss: 0.5324 - val_accuracy: 0.9021
Epoch 34/80
469/469 [==============================] - 16s 35ms/step - loss: 0.5345 - accu
racy: 0.9015 - val_loss: 0.5010 - val_accuracy: 0.9161
Epoch 35/80
469/469 [==============================] - 17s 35ms/step - loss: 0.5317 - accu
racy: 0.9021 - val_loss: 0.5010 - val_accuracy: 0.9148
Epoch 36/80
469/469 [==============================] - 16s 34ms/step - loss: 0.5289 - accu
racy: 0.9028 - val_loss: 0.5029 - val_accuracy: 0.9123
Epoch 37/80
469/469 [==============================] - 15s 32ms/step - loss: 0.5285 - accu
racy: 0.9020 - val_loss: 0.5641 - val_accuracy: 0.8856
Epoch 38/80
469/469 [==============================] - 15s 32ms/step - loss: 0.5265 - accu
racy: 0.9020 - val_loss: 0.4910 - val_accuracy: 0.9143
Epoch 39/80
469/469 [==============================] - 15s 32ms/step - loss: 0.5237 - accu
racy: 0.9024 - val_loss: 0.4834 - val_accuracy: 0.9150
Epoch 40/80
469/469 [==============================] - 15s 33ms/step - loss: 0.5197 - accu
racy: 0.9037 - val_loss: 0.5623 - val_accuracy: 0.8827
```

```
Epoch 41/80
469/469 [==============================] - 16s 35ms/step - loss: 0.5198 - accu
racy: 0.9028 - val_loss: 0.5145 - val_accuracy: 0.8994
Epoch 42/80
469/469 [==============================] - 18s 39ms/step - loss: 0.5175 - accu
racy: 0.9043 - val_loss: 0.5755 - val_accuracy: 0.8787
Epoch 43/80
469/469 [==============================] - 16s 34ms/step - loss: 0.5169 - accu
racy: 0.9034 - val_loss: 0.4775 - val_accuracy: 0.9156
Epoch 44/80
469/469 [==============================] - 16s 35ms/step - loss: 0.5144 - accu
racy: 0.9041 - val_loss: 0.5746 - val_accuracy: 0.8768
Epoch 45/80
469/469 [==============================] - 17s 36ms/step - loss: 0.5132 - accu
racy: 0.9049 - val_loss: 0.6019 - val_accuracy: 0.8778
Epoch 46/80
469/469 [==============================] - 17s 36ms/step - loss: 0.5112 - accu
racy: 0.9051 - val_loss: 0.5325 - val_accuracy: 0.8863
Epoch 47/80
469/469 [==============================] - 17s 36ms/step - loss: 0.5086 - accu
racy: 0.9054 - val_loss: 0.5414 - val_accuracy: 0.8943
Epoch 48/80
469/469 [==============================] - 17s 35ms/step - loss: 0.5078 - accu
racy: 0.9057 - val_loss: 0.5214 - val_accuracy: 0.9051
Epoch 49/80
469/469 [==============================] - 17s 36ms/step - loss: 0.5077 - accu
racy: 0.9044 - val_loss: 0.6438 - val_accuracy: 0.8517
Epoch 50/80
469/469 [==============================] - 16s 34ms/step - loss: 0.5056 - accu
racy: 0.9051 - val_loss: 0.4900 - val_accuracy: 0.9077
Epoch 51/80
469/469 [==============================] - 15s 33ms/step - loss: 0.5052 - accu
racy: 0.9056 - val_loss: 0.5276 - val_accuracy: 0.8889
Epoch 52/80
469/469 [==============================] - 15s 33ms/step - loss: 0.5018 - accu
racy: 0.9074 - val_loss: 0.4692 - val_accuracy: 0.9150
Epoch 53/80
469/469 [==============================] - 15s 32ms/step - loss: 0.5013 - accu
racy: 0.9050 - val_loss: 0.5031 - val_accuracy: 0.9002
Epoch 54/80
469/469 [==============================] - 16s 33ms/step - loss: 0.5002 - accu
racy: 0.9064 - val_loss: 0.4750 - val_accuracy: 0.9141
Epoch 55/80
469/469 [==============================] - 16s 35ms/step - loss: 0.4981 - accu
racy: 0.9054 - val_loss: 0.4581 - val_accuracy: 0.9172
Epoch 56/80
469/469 [==============================] - 17s 35ms/step - loss: 0.5011 - accu
racy: 0.9053 - val_loss: 0.4651 - val_accuracy: 0.9197
Epoch 57/80
469/469 [==============================] - 16s 34ms/step - loss: 0.4953 - accu
racy: 0.9058 - val_loss: 0.5079 - val_accuracy: 0.9058
Epoch 58/80
469/469 [==============================] - 17s 35ms/step - loss: 0.4945 - accu
racy: 0.9059 - val_loss: 0.4658 - val_accuracy: 0.9164
Epoch 59/80
469/469 [==============================] - 17s 36ms/step - loss: 0.4915 - accu
racy: 0.9068 - val_loss: 0.5528 - val_accuracy: 0.8937
Epoch 60/80
469/469 [==============================] - 16s 34ms/step - loss: 0.4933 - accu
racy: 0.9061 - val_loss: 0.4621 - val_accuracy: 0.9138
```

```
Epoch 61/80
469/469 [==============================] - 17s 36ms/step - loss: 0.4925 - accu
racy: 0.9052 - val_loss: 0.4255 - val_accuracy: 0.9276
Epoch 62/80
469/469 [==============================] - 16s 35ms/step - loss: 0.4909 - accu
racy: 0.9059 - val_loss: 0.4451 - val_accuracy: 0.9223
Epoch 63/80
469/469 [==============================] - 16s 34ms/step - loss: 0.4893 - accu
racy: 0.9064 - val_loss: 0.4451 - val_accuracy: 0.9175
Epoch 64/80
469/469 [==============================] - 15s 33ms/step - loss: 0.4892 - accu
racy: 0.9055 - val_loss: 0.4607 - val_accuracy: 0.9127
Epoch 65/80
469/469 [==============================] - 15s 33ms/step - loss: 0.4879 - accu
racy: 0.9062 - val_loss: 0.4654 - val_accuracy: 0.9173
Epoch 66/80
469/469 [==============================] - 15s 33ms/step - loss: 0.4867 - accu
racy: 0.9075 - val_loss: 0.5449 - val_accuracy: 0.8864
Epoch 67/80
469/469 [==============================] - 15s 32ms/step - loss: 0.4867 - accu
racy: 0.9065 - val_loss: 0.5126 - val_accuracy: 0.8899
Epoch 68/80
469/469 [==============================] - 16s 34ms/step - loss: 0.4867 - accu
racy: 0.9070 - val_loss: 0.6011 - val_accuracy: 0.8652
Epoch 69/80
469/469 [==============================] - 16s 34ms/step - loss: 0.4864 - accu
racy: 0.9064 - val_loss: 0.5093 - val_accuracy: 0.8977
Epoch 70/80
469/469 [==============================] - 16s 34ms/step - loss: 0.4834 - accu
racy: 0.9063 - val_loss: 0.5178 - val_accuracy: 0.8986
Epoch 71/80
469/469 [==============================] - 16s 34ms/step - loss: 0.4826 - accu
racy: 0.9063 - val_loss: 0.4685 - val_accuracy: 0.9111
Epoch 72/80
469/469 [==============================] - 16s 34ms/step - loss: 0.4821 - accu
racy: 0.9076 - val_loss: 0.4662 - val_accuracy: 0.9100
Epoch 73/80
469/469 [==============================] - 16s 34ms/step - loss: 0.4818 - accu
racy: 0.9077 - val_loss: 0.5126 - val_accuracy: 0.8938
Epoch 74/80
469/469 [==============================] - 16s 34ms/step - loss: 0.4815 - accu
racy: 0.9062 - val_loss: 0.4612 - val_accuracy: 0.9100
Epoch 75/80
469/469 [==============================] - 15s 33ms/step - loss: 0.4825 - accu
racy: 0.9062 - val_loss: 0.5081 - val_accuracy: 0.8930
Epoch 76/80
469/469 [==============================] - 16s 33ms/step - loss: 0.4807 - accu
racy: 0.9064 - val_loss: 0.4666 - val_accuracy: 0.9075
Epoch 77/80
469/469 [==============================] - 15s 32ms/step - loss: 0.4780 - accu
racy: 0.9064 - val_loss: 0.4854 - val_accuracy: 0.9010
Epoch 78/80
469/469 [==============================] - 16s 33ms/step - loss: 0.4777 - accu
racy: 0.9075 - val_loss: 0.5263 - val_accuracy: 0.8894
Epoch 79/80
469/469 [==============================] - 16s 34ms/step - loss: 0.4754 - accu
racy: 0.9082 - val_loss: 0.4667 - val_accuracy: 0.9115
Epoch 80/80
469/469 [==============================] - 18s 39ms/step - loss: 0.4782 - accu
racy: 0.9072 - val_loss: 0.4719 - val_accuracy: 0.9074
```





> Compare the charts of loss and accuracy for the regularized model with those from training the unregularized model and answer these questions:
>
> 1. Do the charts and numeric metrics for the regularized model show an improvement in the generalization of the model compared to the unregularized model, and why?
> 2. Notice that the test and training loss of the regularized model continue to improve. Do you think that training for additional epochs will be beneficial? **End of exercise.**

> **Answers:**
>
> 1. Yes, the numeric metrics for the regularized model do show an improvement in the generalization of the model compared to the unregularized model. The test loss metric is very stable and close to the train loss. However, Accuracy is not as good as loss. It is true that its trend points to increase, but it still oscilates a lot.

2. I do not think so. Too much training could make the model overfit. I would probably think in a more effective regularization.

Exercise 5-8: The path of the test loss and particularly the test accuracy of the foregoing model is rather erratic. Such behavior often indicates that the gradient is diffcult to estimate and therefore the optimizer exhibits poor convergence. Batch normalization is known to smooth the loss function which improves the ability to consistently estimate gradient. You will now add a **BatchNormalization layer** to the model, along with L2 regularization. Do the following:

1. Start with the model specification you used for Exercise 5-7, change the model name to `nnbr`.
2. After the 512 unit hidden dense layer add a `BatchNormalization` layer, using the argument `momentum=0.998`.
3. To limit the total bias in the model, change the l2 regularization hyperparameter argument for the dense layer to `0.05`.
4. Compile the model.
5. Fit the model for 80 epochs, saving the history object. This will take some time!
6. Plot the training and test loss and accuracy.

In [ ]:
```
## Your code goes here

# 1. Start with the model specification you used for Exercise 5-7, change the
#    model name to nnbr.

nnbr = models.Sequential()
nnbr.add(layers.Dense(28 * 28, activation = "relu", input_shape = (28 * 28, ),


# 3. To limit the total bias in the model, change the l2 regularization
#    hyperparameter argument for the dense layer to 0.05.

nnbr.add(layers.Dense(512, activation = "relu", kernel_regularizer = regularize


# 2. After the 512 unit hidden dense layer add a BatchNormalization layer, usir
#    the argument momentum = 0.998.

nnbr.add(layers.BatchNormalization(momentum = 0.998))


# 4. Compile the model.

nnbr.add(layers.Dense(len(train_labels[0]), activation = "softmax"))
nnbr.compile(optimizer = "rmsprop", loss = 'categorical_crossentropy', metrics


# 5. Fit the model for 80 epochs, saving the history object. This will take sor
```

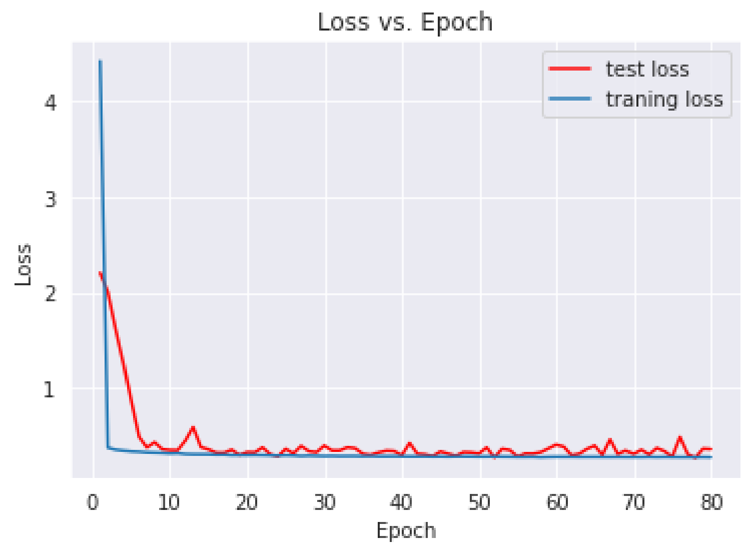
```
history_nnbr = nnbr.fit(train_images, train_labels, batch_size = 128, epochs =


# 6. Plot the training and test loss and accuracy.

plot_loss(history_nnbr)
plot_accuracy(history_nnbr)
```

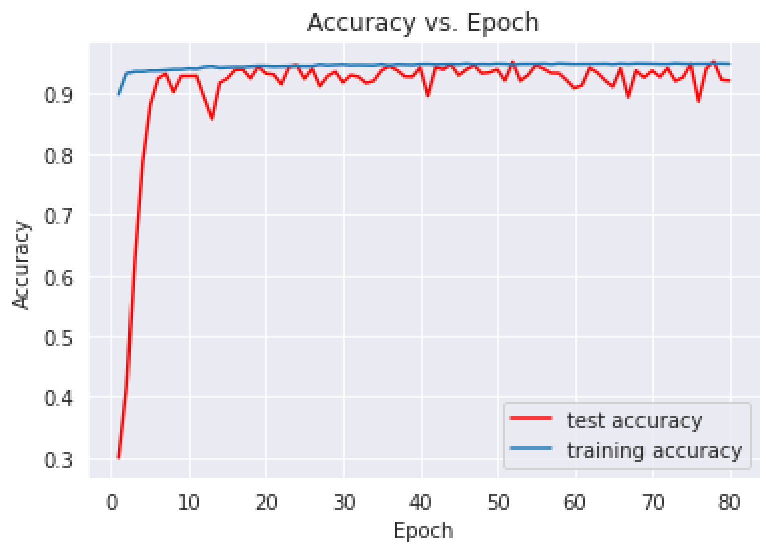
```
Epoch 1/80
469/469 [==============================] - 18s 36ms/step - loss: 4.4132 - accu
racy: 0.8967 - val_loss: 2.2042 - val_accuracy: 0.2995
Epoch 2/80
469/469 [==============================] - 16s 35ms/step - loss: 0.3815 - accu
racy: 0.9317 - val_loss: 2.0044 - val_accuracy: 0.4155
Epoch 3/80
469/469 [==============================] - 16s 35ms/step - loss: 0.3606 - accu
racy: 0.9348 - val_loss: 1.6182 - val_accuracy: 0.6257
Epoch 4/80
469/469 [==============================] - 17s 36ms/step - loss: 0.3513 - accu
racy: 0.9347 - val_loss: 1.2578 - val_accuracy: 0.7860
Epoch 5/80
469/469 [==============================] - 16s 35ms/step - loss: 0.3428 - accu
racy: 0.9362 - val_loss: 0.8579 - val_accuracy: 0.8802
Epoch 6/80
469/469 [==============================] - 16s 35ms/step - loss: 0.3387 - accu
racy: 0.9363 - val_loss: 0.4930 - val_accuracy: 0.9232
Epoch 7/80
469/469 [==============================] - 16s 35ms/step - loss: 0.3323 - accu
racy: 0.9371 - val_loss: 0.3841 - val_accuracy: 0.9311
Epoch 8/80
469/469 [==============================] - 16s 35ms/step - loss: 0.3298 - accu
racy: 0.9383 - val_loss: 0.4385 - val_accuracy: 0.9007
Epoch 9/80
469/469 [==============================] - 17s 36ms/step - loss: 0.3269 - accu
racy: 0.9383 - val_loss: 0.3652 - val_accuracy: 0.9272
Epoch 10/80
469/469 [==============================] - 17s 36ms/step - loss: 0.3216 - accu
racy: 0.9398 - val_loss: 0.3571 - val_accuracy: 0.9271
Epoch 11/80
469/469 [==============================] - 16s 35ms/step - loss: 0.3244 - accu
racy: 0.9390 - val_loss: 0.3554 - val_accuracy: 0.9274
Epoch 12/80
469/469 [==============================] - 16s 34ms/step - loss: 0.3154 - accu
racy: 0.9420 - val_loss: 0.4570 - val_accuracy: 0.8909
Epoch 13/80
469/469 [==============================] - 16s 34ms/step - loss: 0.3132 - accu
racy: 0.9431 - val_loss: 0.5972 - val_accuracy: 0.8565
Epoch 14/80
469/469 [==============================] - 16s 35ms/step - loss: 0.3129 - accu
racy: 0.9408 - val_loss: 0.3875 - val_accuracy: 0.9159
Epoch 15/80
469/469 [==============================] - 16s 34ms/step - loss: 0.3123 - accu
racy: 0.9414 - val_loss: 0.3638 - val_accuracy: 0.9234
Epoch 16/80
469/469 [==============================] - 16s 35ms/step - loss: 0.3090 - accu
racy: 0.9420 - val_loss: 0.3289 - val_accuracy: 0.9376
Epoch 17/80
469/469 [==============================] - 16s 35ms/step - loss: 0.3097 - accu
racy: 0.9418 - val_loss: 0.3277 - val_accuracy: 0.9380
Epoch 18/80
469/469 [==============================] - 16s 34ms/step - loss: 0.3042 - accu
racy: 0.9428 - val_loss: 0.3608 - val_accuracy: 0.9233
Epoch 19/80
469/469 [==============================] - 18s 38ms/step - loss: 0.3053 - accu
racy: 0.9435 - val_loss: 0.3044 - val_accuracy: 0.9430
Epoch 20/80
469/469 [==============================] - 19s 40ms/step - loss: 0.3041 - accu
racy: 0.9435 - val_loss: 0.3355 - val_accuracy: 0.9313
```

```
Epoch 21/80
469/469 [==============================] - 19s 40ms/step - loss: 0.3058 - accu
racy: 0.9427 - val_loss: 0.3330 - val_accuracy: 0.9298
Epoch 22/80
469/469 [==============================] - 17s 35ms/step - loss: 0.3043 - accu
racy: 0.9430 - val_loss: 0.3846 - val_accuracy: 0.9135
Epoch 23/80
469/469 [==============================] - 16s 34ms/step - loss: 0.3020 - accu
racy: 0.9432 - val_loss: 0.3125 - val_accuracy: 0.9425
Epoch 24/80
469/469 [==============================] - 17s 36ms/step - loss: 0.2989 - accu
racy: 0.9436 - val_loss: 0.2915 - val_accuracy: 0.9450
Epoch 25/80
469/469 [==============================] - 17s 37ms/step - loss: 0.3024 - accu
racy: 0.9431 - val_loss: 0.3697 - val_accuracy: 0.9227
Epoch 26/80
469/469 [==============================] - 17s 36ms/step - loss: 0.3001 - accu
racy: 0.9433 - val_loss: 0.3211 - val_accuracy: 0.9396
Epoch 27/80
469/469 [==============================] - 17s 36ms/step - loss: 0.2947 - accu
racy: 0.9460 - val_loss: 0.3997 - val_accuracy: 0.9104
Epoch 28/80
469/469 [==============================] - 17s 36ms/step - loss: 0.2981 - accu
racy: 0.9446 - val_loss: 0.3452 - val_accuracy: 0.9271
Epoch 29/80
469/469 [==============================] - 16s 35ms/step - loss: 0.2959 - accu
racy: 0.9454 - val_loss: 0.3342 - val_accuracy: 0.9345
Epoch 30/80
469/469 [==============================] - 17s 37ms/step - loss: 0.2941 - accu
racy: 0.9459 - val_loss: 0.4032 - val_accuracy: 0.9166
Epoch 31/80
469/469 [==============================] - 17s 37ms/step - loss: 0.2954 - accu
racy: 0.9449 - val_loss: 0.3527 - val_accuracy: 0.9286
Epoch 32/80
469/469 [==============================] - 17s 35ms/step - loss: 0.2935 - accu
racy: 0.9452 - val_loss: 0.3527 - val_accuracy: 0.9260
Epoch 33/80
469/469 [==============================] - 17s 36ms/step - loss: 0.2940 - accu
racy: 0.9450 - val_loss: 0.3855 - val_accuracy: 0.9154
Epoch 34/80
469/469 [==============================] - 17s 36ms/step - loss: 0.2943 - accu
racy: 0.9446 - val_loss: 0.3761 - val_accuracy: 0.9194
Epoch 35/80
469/469 [==============================] - 17s 37ms/step - loss: 0.2924 - accu
racy: 0.9464 - val_loss: 0.3180 - val_accuracy: 0.9366
Epoch 36/80
469/469 [==============================] - 16s 35ms/step - loss: 0.2943 - accu
racy: 0.9444 - val_loss: 0.3056 - val_accuracy: 0.9436
Epoch 37/80
469/469 [==============================] - 17s 35ms/step - loss: 0.2916 - accu
racy: 0.9460 - val_loss: 0.3321 - val_accuracy: 0.9370
Epoch 38/80
469/469 [==============================] - 17s 37ms/step - loss: 0.2915 - accu
racy: 0.9457 - val_loss: 0.3506 - val_accuracy: 0.9265
Epoch 39/80
469/469 [==============================] - 18s 39ms/step - loss: 0.2924 - accu
racy: 0.9453 - val_loss: 0.3465 - val_accuracy: 0.9260
Epoch 40/80
469/469 [==============================] - 17s 37ms/step - loss: 0.2911 - accu
racy: 0.9465 - val_loss: 0.3027 - val_accuracy: 0.9415
```

```
Epoch 41/80
469/469 [==============================] - 17s 36ms/step - loss: 0.2904 - accu
racy: 0.9466 - val_loss: 0.4312 - val_accuracy: 0.8942
Epoch 42/80
469/469 [==============================] - 17s 36ms/step - loss: 0.2902 - accu
racy: 0.9460 - val_loss: 0.3188 - val_accuracy: 0.9419
Epoch 43/80
469/469 [==============================] - 17s 35ms/step - loss: 0.2911 - accu
racy: 0.9462 - val_loss: 0.3117 - val_accuracy: 0.9383
Epoch 44/80
469/469 [==============================] - 17s 36ms/step - loss: 0.2888 - accu
racy: 0.9465 - val_loss: 0.2926 - val_accuracy: 0.9462
Epoch 45/80
469/469 [==============================] - 17s 36ms/step - loss: 0.2898 - accu
racy: 0.9461 - val_loss: 0.3419 - val_accuracy: 0.9284
Epoch 46/80
469/469 [==============================] - 18s 38ms/step - loss: 0.2879 - accu
racy: 0.9474 - val_loss: 0.3194 - val_accuracy: 0.9381
Epoch 47/80
469/469 [==============================] - 18s 38ms/step - loss: 0.2912 - accu
racy: 0.9459 - val_loss: 0.2963 - val_accuracy: 0.9449
Epoch 48/80
469/469 [==============================] - 18s 39ms/step - loss: 0.2904 - accu
racy: 0.9466 - val_loss: 0.3352 - val_accuracy: 0.9316
Epoch 49/80
469/469 [==============================] - 18s 38ms/step - loss: 0.2896 - accu
racy: 0.9462 - val_loss: 0.3320 - val_accuracy: 0.9332
Epoch 50/80
469/469 [==============================] - 17s 36ms/step - loss: 0.2880 - accu
racy: 0.9471 - val_loss: 0.3204 - val_accuracy: 0.9383
Epoch 51/80
469/469 [==============================] - 18s 38ms/step - loss: 0.2889 - accu
racy: 0.9468 - val_loss: 0.3839 - val_accuracy: 0.9196
Epoch 52/80
469/469 [==============================] - 17s 37ms/step - loss: 0.2898 - accu
racy: 0.9454 - val_loss: 0.2781 - val_accuracy: 0.9505
Epoch 53/80
469/469 [==============================] - 17s 37ms/step - loss: 0.2881 - accu
racy: 0.9466 - val_loss: 0.3702 - val_accuracy: 0.9193
Epoch 54/80
469/469 [==============================] - 17s 36ms/step - loss: 0.2874 - accu
racy: 0.9467 - val_loss: 0.3575 - val_accuracy: 0.9289
Epoch 55/80
469/469 [==============================] - 16s 35ms/step - loss: 0.2861 - accu
racy: 0.9468 - val_loss: 0.2881 - val_accuracy: 0.9452
Epoch 56/80
469/469 [==============================] - 17s 36ms/step - loss: 0.2860 - accu
racy: 0.9473 - val_loss: 0.3209 - val_accuracy: 0.9394
Epoch 57/80
469/469 [==============================] - 17s 36ms/step - loss: 0.2867 - accu
racy: 0.9461 - val_loss: 0.3195 - val_accuracy: 0.9322
Epoch 58/80
469/469 [==============================] - 17s 35ms/step - loss: 0.2829 - accu
racy: 0.9477 - val_loss: 0.3342 - val_accuracy: 0.9319
Epoch 59/80
469/469 [==============================] - 16s 35ms/step - loss: 0.2853 - accu
racy: 0.9470 - val_loss: 0.3734 - val_accuracy: 0.9203
Epoch 60/80
469/469 [==============================] - 16s 35ms/step - loss: 0.2868 - accu
racy: 0.9464 - val_loss: 0.4134 - val_accuracy: 0.9072
```

```
Epoch 61/80
469/469 [==============================] - 16s 34ms/step - loss: 0.2866 - accu
racy: 0.9466 - val_loss: 0.3920 - val_accuracy: 0.9119
Epoch 62/80
469/469 [==============================] - 16s 35ms/step - loss: 0.2866 - accu
racy: 0.9466 - val_loss: 0.3027 - val_accuracy: 0.9409
Epoch 63/80
469/469 [==============================] - 17s 35ms/step - loss: 0.2855 - accu
racy: 0.9468 - val_loss: 0.3180 - val_accuracy: 0.9328
Epoch 64/80
469/469 [==============================] - 16s 35ms/step - loss: 0.2857 - accu
racy: 0.9471 - val_loss: 0.3685 - val_accuracy: 0.9199
Epoch 65/80
469/469 [==============================] - 16s 35ms/step - loss: 0.2861 - accu
racy: 0.9462 - val_loss: 0.4034 - val_accuracy: 0.9096
Epoch 66/80
469/469 [==============================] - 17s 37ms/step - loss: 0.2853 - accu
racy: 0.9475 - val_loss: 0.3060 - val_accuracy: 0.9400
Epoch 67/80
469/469 [==============================] - 17s 35ms/step - loss: 0.2852 - accu
racy: 0.9467 - val_loss: 0.4674 - val_accuracy: 0.8923
Epoch 68/80
469/469 [==============================] - 16s 35ms/step - loss: 0.2856 - accu
racy: 0.9475 - val_loss: 0.3150 - val_accuracy: 0.9365
Epoch 69/80
469/469 [==============================] - 17s 37ms/step - loss: 0.2848 - accu
racy: 0.9474 - val_loss: 0.3521 - val_accuracy: 0.9251
Epoch 70/80
469/469 [==============================] - 17s 36ms/step - loss: 0.2843 - accu
racy: 0.9471 - val_loss: 0.3157 - val_accuracy: 0.9362
Epoch 71/80
469/469 [==============================] - 17s 36ms/step - loss: 0.2842 - accu
racy: 0.9466 - val_loss: 0.3602 - val_accuracy: 0.9256
Epoch 72/80
469/469 [==============================] - 17s 35ms/step - loss: 0.2859 - accu
racy: 0.9466 - val_loss: 0.3102 - val_accuracy: 0.9403
Epoch 73/80
469/469 [==============================] - 17s 37ms/step - loss: 0.2838 - accu
racy: 0.9477 - val_loss: 0.3783 - val_accuracy: 0.9188
Epoch 74/80
469/469 [==============================] - 17s 37ms/step - loss: 0.2855 - accu
racy: 0.9471 - val_loss: 0.3447 - val_accuracy: 0.9251
Epoch 75/80
469/469 [==============================] - 17s 37ms/step - loss: 0.2846 - accu
racy: 0.9467 - val_loss: 0.2860 - val_accuracy: 0.9472
Epoch 76/80
469/469 [==============================] - 17s 37ms/step - loss: 0.2833 - accu
racy: 0.9470 - val_loss: 0.4944 - val_accuracy: 0.8853
Epoch 77/80
469/469 [==============================] - 17s 36ms/step - loss: 0.2834 - accu
racy: 0.9470 - val_loss: 0.3144 - val_accuracy: 0.9395
Epoch 78/80
469/469 [==============================] - 16s 35ms/step - loss: 0.2829 - accu
racy: 0.9473 - val_loss: 0.2759 - val_accuracy: 0.9511
Epoch 79/80
469/469 [==============================] - 16s 35ms/step - loss: 0.2842 - accu
racy: 0.9474 - val_loss: 0.3720 - val_accuracy: 0.9210
Epoch 80/80
469/469 [==============================] - 16s 34ms/step - loss: 0.2825 - accu
racy: 0.9470 - val_loss: 0.3669 - val_accuracy: 0.9197
```





> Compare the graphs to the train and test loss and train and test accuracy to the foregoing model without the batch normalization. What does the difference in smoothness, especially for the test accuracy, tell you about the change in behavior of the gradient?

> **Answer:**
>
> nnbr model has a much better behavior compared with unregularized model. Now, both accuracy and loss have some variability but its ceiling (or floor) is the results for training, which makes total sense. Not only accuracy is smoother but also loss has little variability. Since both values oscilate within a small range, this results indicate that the change in behavior of the gradient is mostly uniform solving the problem with good generalization.

```
In [ ]:
```