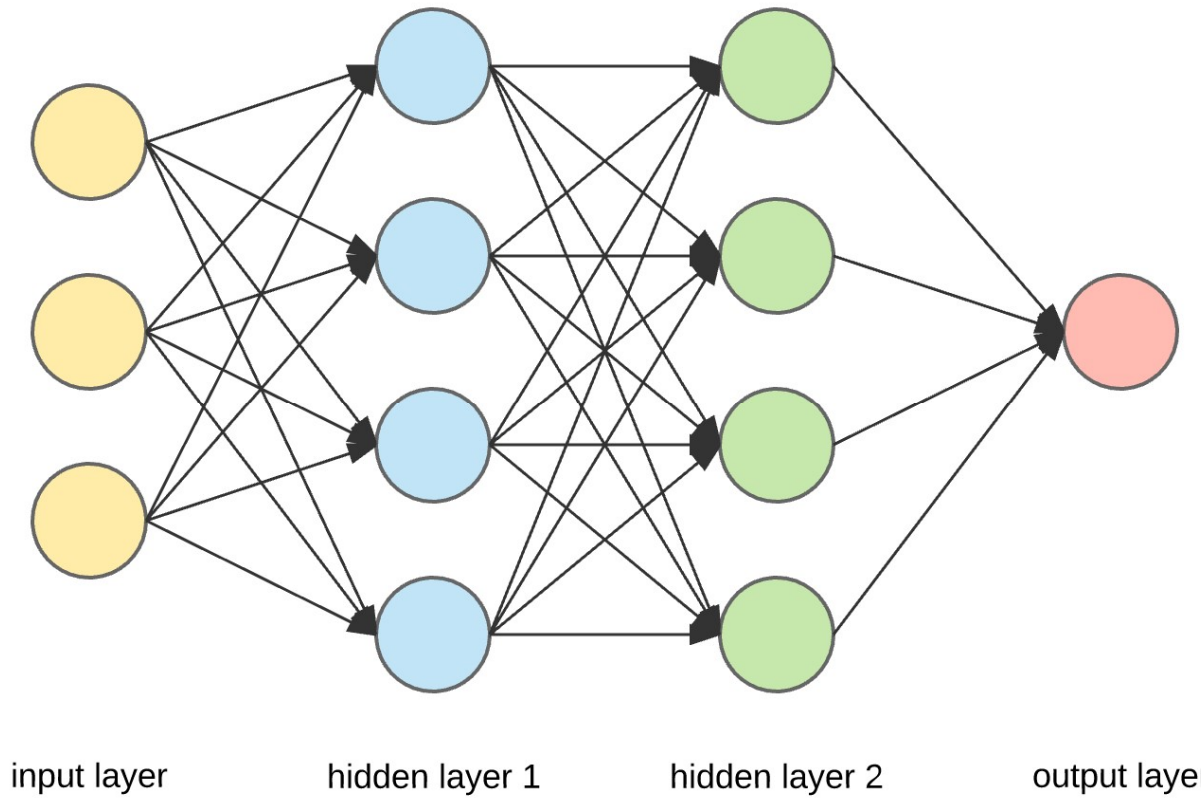
Neural Networks: Backpropagation and Deep Learning

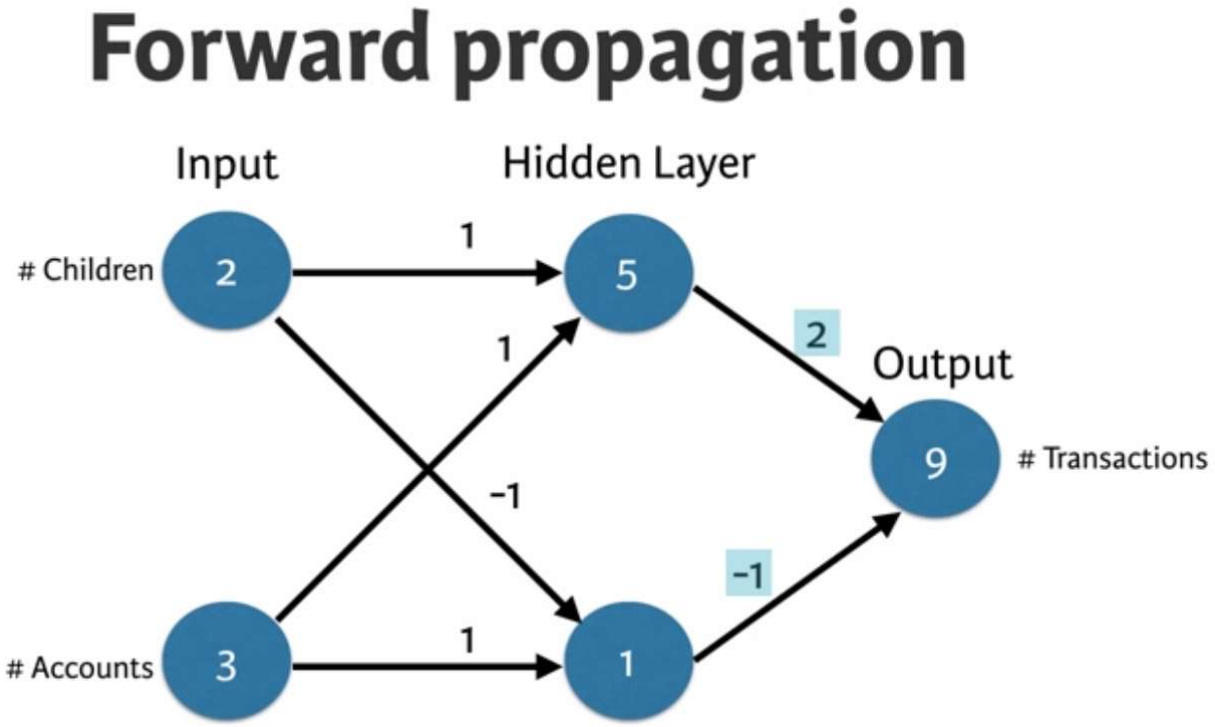
Simple linear regression models don't allow for us to consider interactions between feature variables. However, in generalized linear regression we can include interactions terms as new features, usually of product of said features, and then we can estimate the appropriate weight or regression coefficient on these parameters. However, the representation of the interaction might not be entirely natural or encompass how several features collectively impact a target variable. Much like general linear model's neural network can measure the effect of interactions on target variables and do so particularly well. Since neural networks no longer restrict us to linear combinations, or weights in this case, we can represent interactions far more generally by imbedding them into the complex webbings of deep neural networks.

A deep neural network is simply a neural network with many hidden layers. To put it bluntly neural networks are a function with weights as parameters, features as imput variables, and the estimated target variable as its output. Our goal is to optimize the network with best weights we can find so that our estimated target is as close to the actual target as we can make it. Now, this view brushes over the interworking's of the neural network. In a regression task, we start with our feature variables and feed these into the network as node values in the first layer, for now we assume the network is already parameterized with weights, we then calculate the each node value in the next layer by dotting our node vector in the input layer with respect to the column of the weight matrix of the node we wish to calculate. That is if we want to calculate the

2^{nd} node value in the 2^{nd} layer we dot the input vector(our feature data) with a vector of weights that connect each node in the input layer to the 2^{nd} node in the next layer. Then, we take the resulting scalar and apply an activation function to it. Historically this function was a sigmoid function which squashed each node to value between zero and one, now though we use something called ReLU which is defined to be max(0, a) where a is the resulting dot product mentioned before. We won't delve into why these functions are used and why one is used now over the other, but it is important to note the usage of both just in case. We continue this process for each node in each layer of the network, this process is referred to as forward propagation. The intermediary layers between the input and output layer are referred to as hidden layers, it is here that the network is able to pick up on complex interactions in the data.



We will explore some pedagogical examples of forward propagating a neural network. The use of code will continue throughout the paper, most of these examples are simplified for conceptual ease. Let us consider a dataset containing data on bank customers. Say that we have data on the number of accounts, children, and monthly transactions for each individual and we want to predict how many transactions a customer makes each month using the number of children and accounts as our feature variables. Consider the network as follows:



This is an example of forward propagation, with the weights already given, we predict that an individual with 2 children and 3 accounts will make 9 monthly transactions. Note that the hidden layer is calculated exactly as described above. Lets look at the code for this:

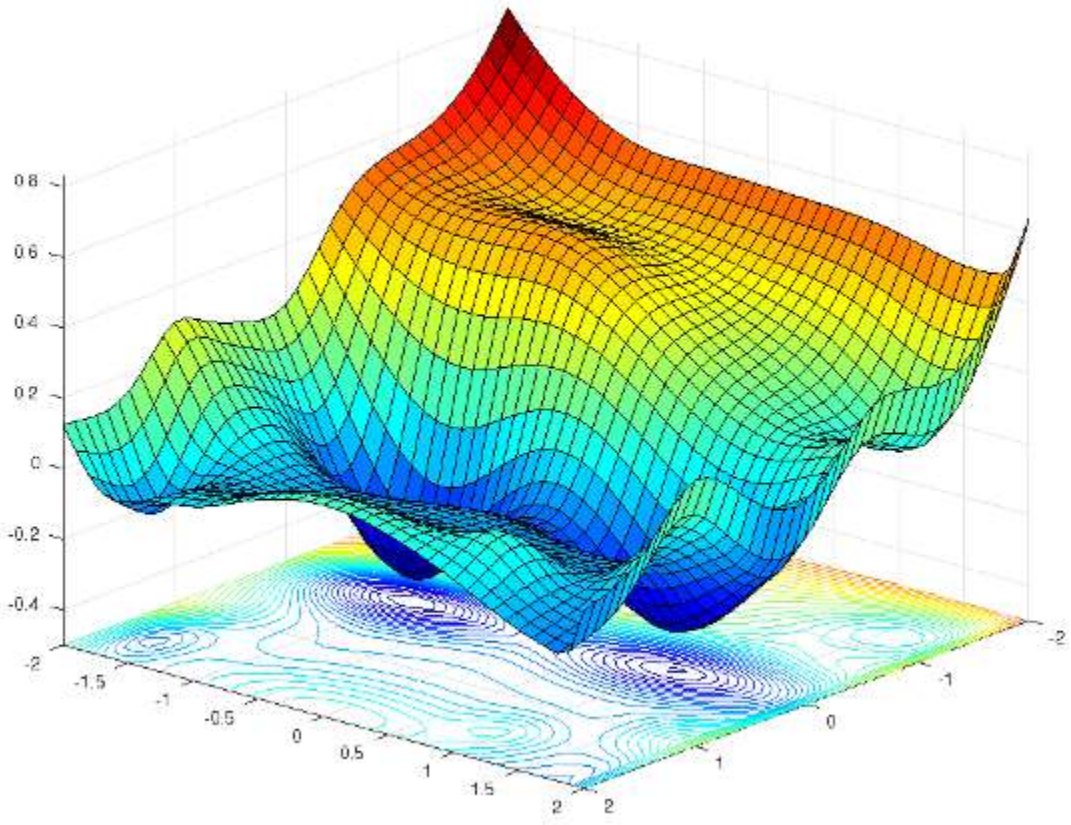
```
import numpy as np
def relu(input):
    output = max(0, input)
    return(output)

input_data = np.array([2,3])
weights = {'node_0':np.array([1,1]),
           'node_1':np.array([-1,1]),
           'output':np.array([2,-1])}

node_0_input = (input_data * weights['node_0']).sum()
node_0_output = relu(node_0_input)
node_1_input = (input_data * weights['node_1']).sum()
node_1_output = relu(node_1_input)
hidden_layer_outputs = np.array([node_0_output, node_1_output])
print(hidden_layer_outputs)

>>> [5 1]
```

Similarly, to calculate the output we run:

```
model_output = (hidden_layer_outputs * weights['output']).sum()
print(model_output)

>>> 9
```

You may have noticed that in this example the weights, the ones we wish to optimize, are already given. However, this is usually the case, we tend to start with some weights often chosen at random and then after forward propagation, we check to see if those weights were any good at predicting the actual target. Consider the same example, but now suppose that the actual number of monthly transactions for that individual is 13. Recall that we predicted a value of 9, meaning we made an error of 4. In a more general problem we compute the error of our predictions using the SSE or MSE metric, in order to optimize the weights we use an algorithm called gradient descent. Gradient descent looks at the error as a function of the weights and attempts to look for a

minimum value, thus minimizing the error of model and parameterizing the optimal network. But, finding a global minimum is nearly impossible in more complex problems. Consider for example

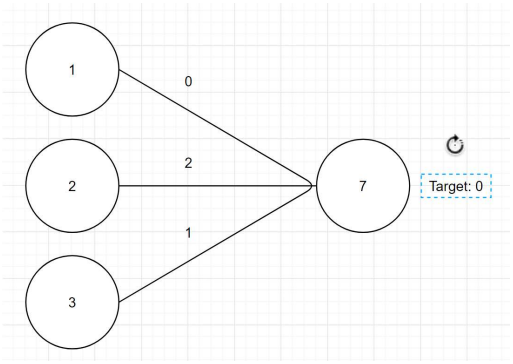


There are two local minima, and as one might imagine solving for the absolute minima of surface this complex would be quite tedious. Imagine the increase of difficulty for a surface in R^p.

Gradient descent is implemented alongside backpropagation. Backpropagation is performed by first parameterizing weights randomly, forward propagating, calculating the error, and then applying gradient descent. Gradient descent relies on the chain rule form calculus. Here we will focus more on concepts instead of the mathematical details.

We start by nudging the weights in value, then we observe how this nudge affects the pre-activated node values, the node value themselves, and finally the error. If our nudge decreased the error we go 'backwards' through the network updating each weight. Then we forward propagate and repeat until we settle upon a minimum error value.

Now, some details are in order. These so called 'nudges' can overshoot and miss potential minima so we attach a coefficient called a 'learning' rate. This value is generally small(0.01) and helps ensure that we avoid going to far astray in our search of a minima. There are plenty of libraries that implement gradient descent effectively, one such is stochastic bath gradient descent, which introduces a random component to the weight changes and helps the algorithm to converge to a minimum with fewer backpropagation iterations. We recommend the use of these algorithms. In some simple cases we can use results from calculate the gradient. One particularly useful tool is that ReLU has a either a slope of zero or one, reducing our problem to a product of the pre-activated node value slope w.r.t the weight being updated, the slope of the error function w.r.t the node value, and the learning rate. Consider the following network.



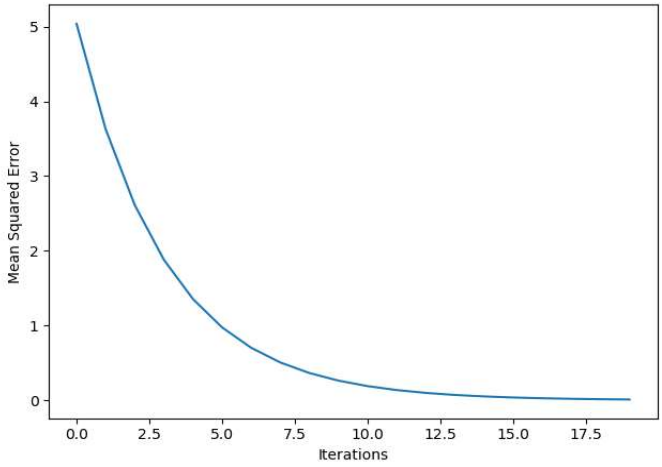
We implement this in python as follows:

```
input_data = np.array([1,2,3])
weights = np.array([0,2,1])
target = 0
preds = (weights * input_data).sum()
error = preds - target
gradient = 2 * input_data * error
learning_rate = 0.01
weights_updated = weights - learning_rate * gradient
preds_updated = (weights_updated * input_data).sum()
error_updated = preds_updated - target
print(error)
print(error_updated)

>>> 7
>>> 5.04
```

As you can see the error decreased! If we continue backpropagating we can continue to reduce error until the returns are very small and not worth reiterating.

```
from matplotlib import pyplot as plt
def get_slope(input_data, target, weights):
    preds = (weights * input_data).sum()
    error = preds - target
    return(2 * input_data * error)
def get_mse(input_data, target, weights):
    preds = (weights * input_data).sum()
    return(abs(preds - target))


n_updates = 20
mse_hist = []

# Iterate over the number of updates
for i in range(n_updates):
    # Calculate the slope: slope
    slope = get_slope(input_data, target, weights)

    # Update the weights: weights
    weights = weights - 0.01 * slope

    # Calculate mse with new weights: mse
    mse = get_mse(input_data, target, weights)

    # Append the mse to mse_hist
```

```
    mse_hist.append(mse)

# Plot the mse history
plt.plot(mse_hist)
plt.xlabel('Iterations')
plt.ylabel('Mean Squared Error')
plt.show()
```



Creating a keras model:

Understanding backpropagation allows us to truly explore deep learning problems and apply useful tools for deep learning and backpropagation. We will use the keras library which has a particularly rich library of deep learing tools and techniques. Keras is built directly off of TensorFlow and utilizes a terse and approachable syntax as compared to TensorFlow, this will be particularly useful in familiarizing ourselves with some of the TensorFlow tools without overwhelming ourselves with customization options and so forth. Using the keras library we will create and optimize networks using the keras interface to the TensorFlow library. We will start by specifying a model

```
### import libraries for building the model
import numpy as np
from keras.layers import Dense
from keras.models import Sequential
```

```
### Read in data and specify number nodes in first layer
predictors = np.loadtxt('predictorss_data.csv', delimeter=',')
n_cols = predictors.shape[1]

### Specify the model type
model = Sequential()
model.add(Dense(100, activation='relu', input_shape =(n_cols,)))
model.add(Dense(1))
```

A sequential model requires that each layer has weights or connections on to the one layer coming directly after in the network. A dense layer is layer such that every node in the current layer connects to every node in the previous layer. In each layer we specify the number of nodes in each layer. Notice then, the last layer has one node, the output node. Thus, we are looking a network with an input layer, one hidden layer, and an output layer.

After we have setup a model, we will need to compile it. That is, ready the network for optimization, or rather backpropagation. Compiling usually entails setting the learning rate and specifying and algorithm for optimization. Much of this is subjective, there is not necessarily a 'best' way to optimize. A popular optimizer is "Adam", which adjusts the so called learning rate as it does gradient descent. We will also specify the loss function, those who have studied statistics will know that this metric can greatly impact results, and those who have studied analysis will know that different models will only converge to minimums under certain metrics. To optimize in Keras we would build a model just like we did above and add the following code:

```
model.compile(optimizer ='adam', loss ='mean_squared_error')
```

After compiling we can fit our model! This step involves actually applying backpropagation with gradient descent to the network and updating to the optimal weights. We can perform a scaling operation on our data, that is, a transform of our feature variables that effectively normalize and/or center the data to similar values. Scaling may ease the optimization process and thus making fitting the model easier. To fit the model we run:

```
model.fit(predictors, target)
```

Classification Tasks with Deep Learning

Thus far we've only done regression tasks with deep learning, however deep learning applies just as well to classification tasks. Now, with classification tasks we will have to make some assumptions in order to talk about something like a loss function, furthermore the usual loss functions from regression will no longer apply. We will use a common loss function in classification problems called *categorical cross entropy.* Cross entropy comes from information theory, where "the cross entropy between two probability distributions *p* and *q* over the same underlying set of events measures the average number of bits need to identify and event drawn from the set, if a coding scheme is used that is optimized for an 'unnatural' probability distribution *q*, rather than the 'true' distribution *p*."(Wikipedia) To summarize, cross entropy is a tool that allows us to crudely quantify categorical data and take metrics on that data as if it were

numerical, whilst also acknowledging the unnaturalness of this process and adjusting where possible.

We will disregard the ReLu activation function in favor for the *softmax* activation function. The *softmax* is a logistic function, the class of functions as the well known sigmoid, that effectively transforms a K-dimensional vector of numbers so that each entry is between 0 and 1 and also ensures that summing the resulting components of said vector produces a value of exactly 1. In statistics this approach is used to interpret or estimate probabilities, much like logistic regression. The code to run a classifaction task would emulate the following:

```
import pandas as pd
from keras.layers import Dense
from keras.models import Sequential
from keras.utils import to_categorical
data = pd.read_csv("basketball_shot_log.csv")
predictors = data.drop(['shot_result'], axis=1).matrix()
n_cols = predictors.shape[1]
target = to_categorical(data.shot_result)
model = Sequential()
model.add(Dense(100, activation='relu', input_shape =
(n_cols,)))
model.add(Dense(100,activation='relu'))
model.add(Dense(100,activation='relu'))
model.add(Dense(2, activation= 'softmax'))
model.compile(optimizer = 'adam',
              loss ='categorical_crossentropy', metrics =
['accuracy'])
model.fit(predictors,target)
```

Lastly, don’t forget to save the model!!!

```
from keras.models import load_model
model.save('model_file.h5')
my_model = load_model('my_model.h5')
predictions = my_model.predict(data_to_predict_with)
probability_true = predictions[:,1]
```

Conclusion

This paper aimed to introduce the concepts of deep-learning and deep neural networks while also familiarizing the read with code in python for tackling various deep learning methods. We have yet to study model optimization and choosing model architecture. We will explore how to do this in the next paper in the keras library as well as delve into the larger TensorFlow library.

References

All code was taken from exercises provided by Data Camp in the following

Becker, Dan. "Introduction to Deep Learning | Python." *R*, campus.datacamp.com/courses/deep-learning-in-python/basics-of-deep-learning-and-neural-networks?ex=1.

"Gradient Descent." *Wikipedia*, Wikimedia Foundation, 15 May 2018, en.wikipedia.org/wiki/Gradient_descent.

Kolkiewicz, Adam. "Stochastic Mesh Method." *Encyclopedia of Quantitative Finance*, 2010, doi:10.1002/9780470061602.eqf13013.

"Optimizers." *Keras Documentation*, keras.io/optimizers/.

Raschka, Sebastian. *Python Machine Learning*. Packt, 2015.

"Softmax Function." *Wikipedia*, Wikimedia Foundation, 3 May 2018, en.wikipedia.org/wiki/Softmax_function.

"What Is Cross-Entropy, and Why Use It?" *Open Letter to Professor Sung-Chul Shin, President of KAIST*, www.cse.unsw.edu.au/~billw/cs9444/crossentropy.html.