University of Strathclyde Glasgow

CS310 – AI Foundations

Andrew Abel

March 2024

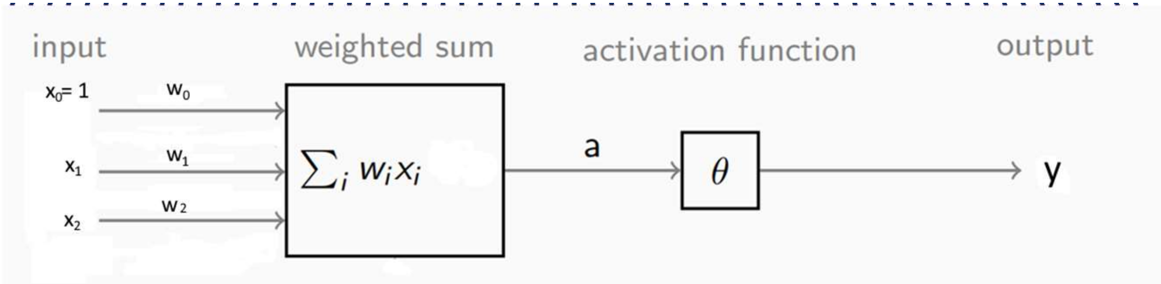
Week 8: Real Network Modelling

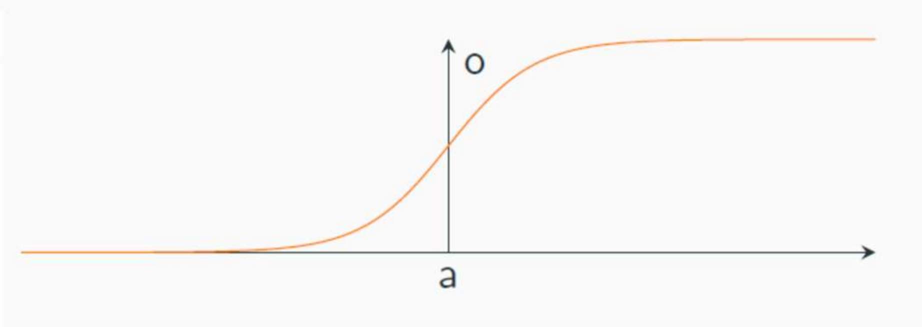
Welcome!

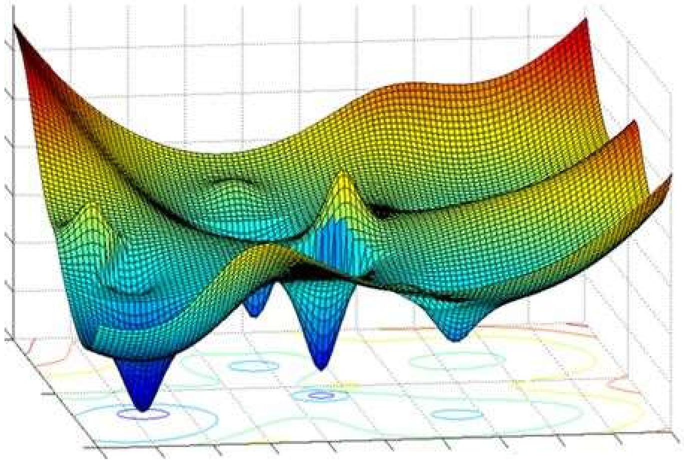
- Previous topic was very heavy!
  - A lot of gradient descent, backpropagation etc.
  - Now we will introduce the bigger picture
- Tensorflow
- Creating a model

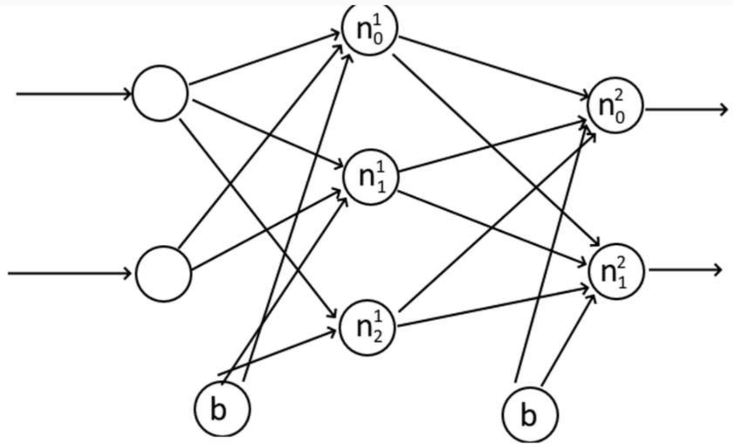
Previously

- We had our perceptron
- Went from binary step to sigmoid activation
- Built a fully connected feedforward neural network
- Performed back propagation with gradient descent







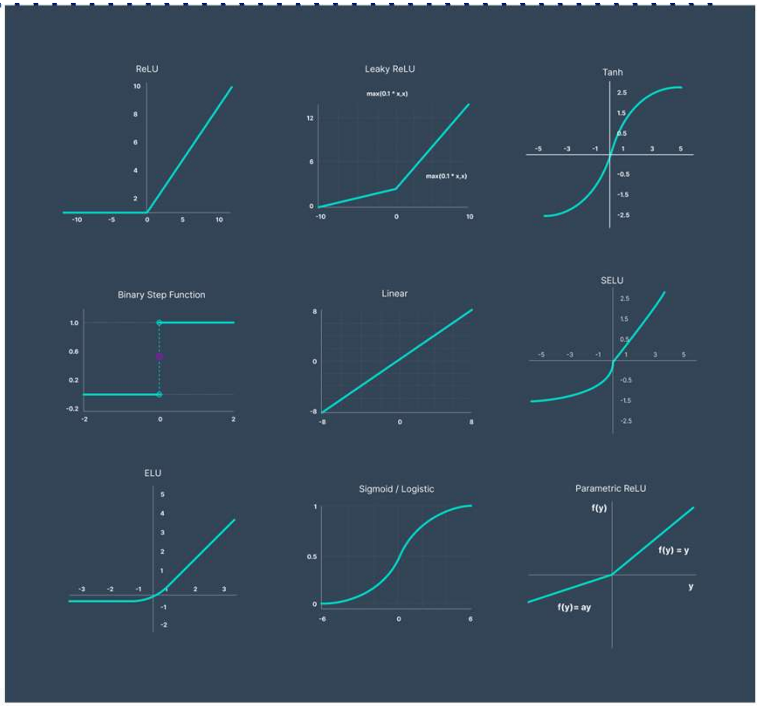


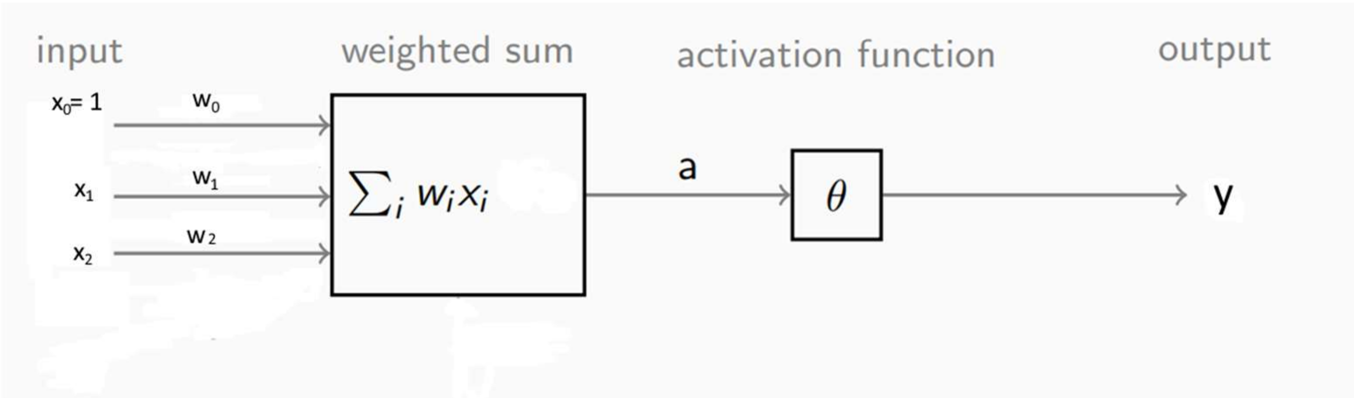
Is this the state-of-the-art?

- Much of that is because it is good for teaching
  - Sigmoid is easily differentiable manually
- Back propagation is the “classic” approach
  - Still in use today, but many variations!
- A lot of different activation functions
  - Sigmoid is no longer the recommended approach
- There are even different types of neurons

Activation Upgrades

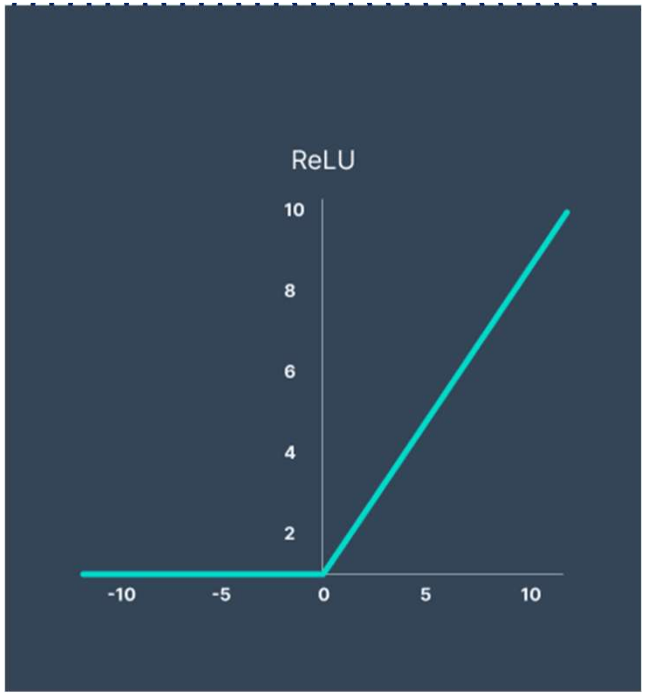
- Still widely used!
- However, binary step and sigmoid are not so widely used
- Many different activation functions advised
- Discussed in previous lecture
- https://www.v7labs.com/blog/neural-networks-activation-functions





ReLU

- Rectified Linear Unit
- f(x) = max(0,x)
- It looks linear, but does have a derivative
- Currently very fashionable
- Recommended for use in hidden layers by default
- Does not activate for negative values
  - More efficient
  - Can mean some weights and inputs are never trained (dead neurons)

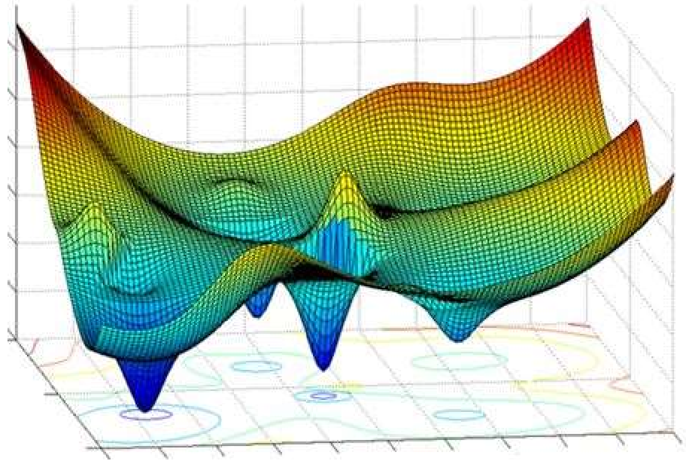


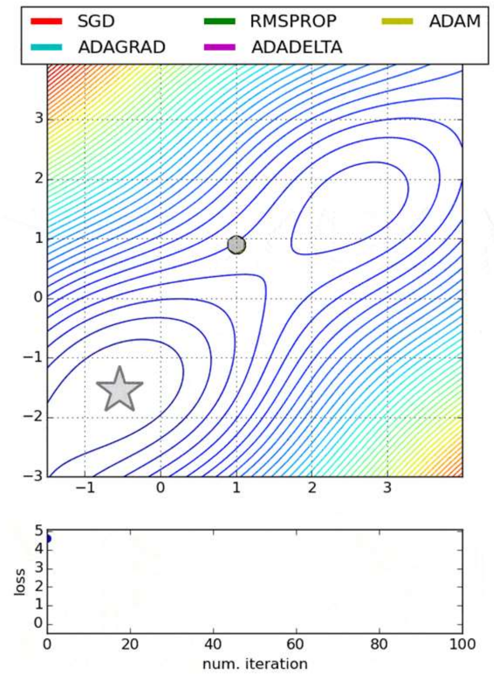
Activation Function Recommendations

- Hidden layers recommendations
  - **Convolutional Neural Network (CNN)**: ReLU activation function.
  - **Recurrent Neural Network**: Tanh and/or Sigmoid activation function.
- Output layer recommendations
  - **Regression** - Linear Activation Function
  - **Binary Classification** - Sigmoid/Logistic Activation Function
  - **Multiclass Classification** - Softmax
  - **Multilabel Classification** - Sigmoid
- Trial and error works too!

Training and Optimisation

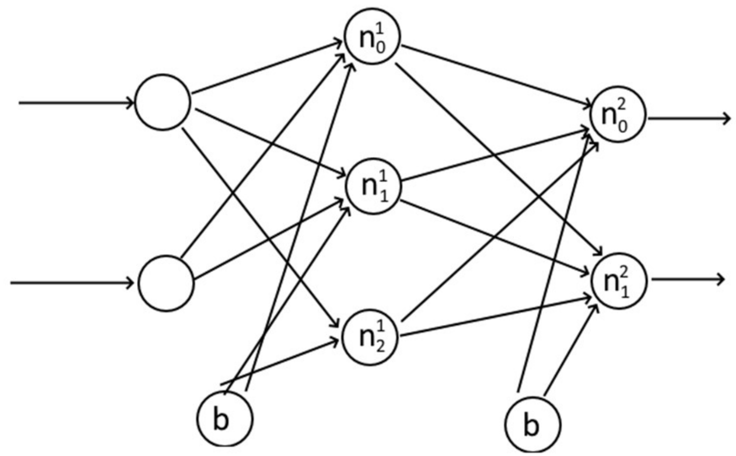
- Gradient Descent was the “standard”
- But many variations exist
    - Momentum, Adam etc.
- One of the mostly widely used is **Adam (Adaptive Moment Estimation)**
    - Generally a good choice!
- Not going to discuss here
- https://towardsdatascience.com/optimizers-for-training-neural-network-59450d71caf6





Fully Connected Networks

- Still extremely valuable and current
- Fully connected layers are known as dense layers
- These take an input, and feed it forward
- Most models will include at least 1 fully connected layer



Tensorflow for Machine Learning in Python

Building Your Own

- Python is a recommended language for machine learning
- There are two main libraries Tensorflow and Pytorch
  - Other libraries are available
- Tensorflow is enormous, we are not going to go through it all!
  - https://www.tensorflow.org/
- We will walk through creating a very simple network

Creating your Environment

- You will likely need to add several packages to your Python install
- tensorflow – the machine learning package
- pandas – Python Data Analysis package
- sklearn – Another machine learning package
- Keras – Python interface for machine learning
    - now effectively tied together with Tensorflow
- If running this on a laptop, you WILL get error/warning messages, this is designed for GPU processing, not CPU processing, but it will still work

```
# check version
import tensorflow
print(tensorflow.__version__)
```

Stages of creating a machine learning model

1. Prepare Dataset
2. Define the model.
   - Select layers, activations, etc.
3. Compile the model.
   - Creates the model
4. Fit the model.
   - Training process
5. Evaluate the model.
   - Test training performance
6. Make predictions.

Prepare Dataset

- Sign up for Kaggle
  - https://www.kaggle.com/datasets?tags=13207-Computer+Vision
  - One of the best resources for datasets
- Split into Training and Test sets (if not already done)
- You may want to normalise
- Read into Python
- Store as a pickle
  - Research this!

```
import numpy as np
data_path = "./"
train_data = np.loadtxt(data_path + "mnist_train.csv", delimiter=",")
test_data = np.loadtxt(data_path + "mnist_test.csv", delimiter=",")
```

Prepare Dataset II

- You may need to do some work to identify labels
- Also, if just one dataset, you may want to split into training and test sets

```
X_train, X_test, y_train, y_test
                  = train_test_split(X, y, test_size=0.33)
```

  - Not needed for MNIST dataset
  - Can use train_test_split sklearn function
  - Read_csv function may also be useful

- Do similar work to last week, define training and test sets and labels
  - Do not need bias
  - Might want to define your lists as numpy arrays, then we can use "shape"

```
train_input = np.array([np.array(d[1:]) for d in train_data ])
```

Define the Model

- Select model you need, then choose architecture
  - define the layers of the model
  - configure each layer with a number of nodes and activation functions
  - connect the layers together into a cohesive model.
  - Tensorflow API useful
    - https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dense
  - Also the guide
    - https://machinelearningmastery.com/tensorflow-tutorial-deep-learning-with-tf-keras/

```
# groups models together into a sequential model
model = Sequential()
```

Define the Model

- Example model, create a sequential model (the simplest type)
- We create a fully connected hidden layer which receives input from our input (so 784 features
  - Bias is added automatically
  - Has 4 neurons
  - Activation functions are sigmoid shaped
- A second layer is created, and in this case we use 10 neurons (i.e. one for each number we are trying to predict)
  - We use sigmoid again, but ‘softmax’ might be better
- Can create as many layers as we want
  - The guide helps with options!
  - https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dense

```
#  Step 2, define model
model = Sequential()
model.add(Dense(4, activation='sigmoid', input_shape=(n_features,)))
model.add(Dense(10, activation='sigmoid'))
```

Compile the Model

- Compiling the model requires that you first select a loss function that you want to optimize, such as mean squared error or cross-entropy.
- Requires that you select an algorithm to perform the optimization procedure
  - Can use stochastic gradient descent
  - more modern variations, such as Adam are available
- The optimizer can be specified as a string for a known optimizer class, e.g. '*sgd*' for stochastic gradient descent, or you can configure an instance of an optimizer class and use that.
  - Note here, can define learning rate and momentum (additional property)

```
# compile the model
opt = SGD(learning_rate=0.01, momentum=0.9)
model.compile(optimizer=opt, loss='binary_crossentropy')
```

Compile the Model

- The three most common loss functions:
  - ‘*binary_crossentropy*‘ for binary classification.
  - ‘*sparse_categorical_crossentropy*‘ for multi-class classification.
  - ‘*mse*‘ (mean squared error) for regression.
- More here
  - https://www.tensorflow.org/api_docs/python/tf/keras/optimizers
- Here we use sgd
- Test the loss with crossentropy (I had problems with MSE for this problem!)
- Use accuracy as our measure

```
#  Step 3, compile model
model.compile(optimizer='sgd', loss='SparseCategoricalCrossentropy',
metrics=['accuracy'])
model.summary()
```

Compile the model

- A number of approaches you can use
  - Experiment and see!
- https://analyticsindiamag.com/guide-to-tensorflow-keras-optimizers/
- https://python.plainenglish.io/mastering-optimizers-with-tensorflow-a-deep-dive-into-efficient-model-training-81c58c630ef1

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

model.compile(optimizer='sgd', loss='SparseCategoricalCrossentropy’,
        metrics=['accuracy’])

opt = SGD(learning_rate=0.01, momentum=0.9)model.compile(optimizer=opt,
        loss='binary_crossentropy’)
model.compile(optimizer=opt, loss='binary_crossentropy')
```

Fit the Model

- ‘Fitting’ refers to training
  - ‘Fit’ the model to the data

```
model.fit(train_input, train_label, epochs=5, batch_size=32,
verbose=1)
```

- Need to select the training config
  - Number of epochs (number of times to train)
  - Batch size (samples used to estimate the error)
- Training tries to minimise the loss function, and backpropagates the model
- Slowest part of machine learning. Can take hours, days, even weeks!
- By setting “verbose” to 1, can see progress
  - Set to 0 to get no display

Fit the Model

- Slowest part of machine learning. Can take hours, days, even weeks!
- More epochs means it might fit the model better, but might cause **overfitting**
- By setting “verbose” to 1, can see progress
  - Set to 0 to get no display

```
Epoch 1/5
1875/1875 [==============================] - 1s 709us/step - loss: 1.9657 - accuracy: 0.2945
Epoch 2/5
1875/1875 [==============================] - 1s 613us/step - loss: 1.7178 - accuracy: 0.3834
Epoch 3/5
1875/1875 [==============================] - 1s 598us/step - loss: 1.6049 - accuracy: 0.3994
Epoch 4/5
1875/1875 [==============================] - 1s 613us/step - loss: 1.5169 - accuracy: 0.4136
Epoch 5/5
1875/1875 [==============================] - 1s 598us/step - loss: 1.4732 - accuracy: 0.4106
```

Evaluate the Model

- You should use separate data for testing your model
  - we should get an unbiased estimate of the performance of the model when making predictions on new data.
- Much faster than training, as you do not need to adjust the weights
- From an API perspective, this involves calling a function with the holdout dataset and getting a loss and perhaps other metrics that can be reported.

```
# Step 5, evaluate the model
loss, acc = model.evaluate(test_input, test_label, verbose=0)
print('Test Accuracy: %.3f' % acc)
```

Predictions!

- The fun part of the model
  - How well can it classify your input?
- For this one, we are looking to see how well it classifies, so which one of the 10 cases
- Other problems may want to see a different output

```
print("actual: ",test_label[22])
row = np.array(test_input[22])[None,...]
yhat = model.predict(row)
print('Predicted: %s (class=%d)' % (yhat,
np.argmax(yhat)))
fig = plt.figure()
plt.plot(yhat[0])
plt.show()
```

Other tips

- When working with this code, it can be easy to mess up configurations
- Create a separate virtual environment (venv)
  - This gives you a clean environment for you to install packages
  - Doesn't install them elsewhere
- If you do not have a GPU, then you may get warning messages about using tensorflow
  - It will still work, but a little slower
- A lot of the work will be trial and error. Experiment with different configurations

And that is your first model!

- You should improve on this for your week 9 lab
- Experiment with settings
  - Reading input data
  - Creating dense (fully connected layers)
  - Generating output
  - Testing