Summary Notes

- **NN-1: Intro to NN**

  What are the issues with Classic ML Algos?

  - Require us to do manual feature engineering to be able to create complex boundaries
  - They might not always work well with big datasets, and we are living in a big-data regime
  - Works poorly for sparse data and for unstructured data (image/ text/ speech data)

  Clearly, all these algos have some limitations which call for a more powerful ML model that can work in the conditions mentioned above. This brings us to Neural Networks (NN).

  NN models power pretty much every minute of your digital life

  - Online Ads (Google, YouTube)
  - Data compression: Done using Autoencoders
  - Image enhancement: Eg Magic Eraser
  - Gmail: Eg Autocomplete, smart reply

  Where did the inspiration for a neuron come from?

  NN is loosely inspired by the biological neurons found in the human brain.

  In the brain, there exist biological neurons that are connected to each other, forming a network

  In simple terms, we understood that

  - Neuron takes input(s)
  - Perform some computations.
  - Ultimately, it fires/passes the output to further neurons, for further processing.

  How does an Artificial neuron do its computations?

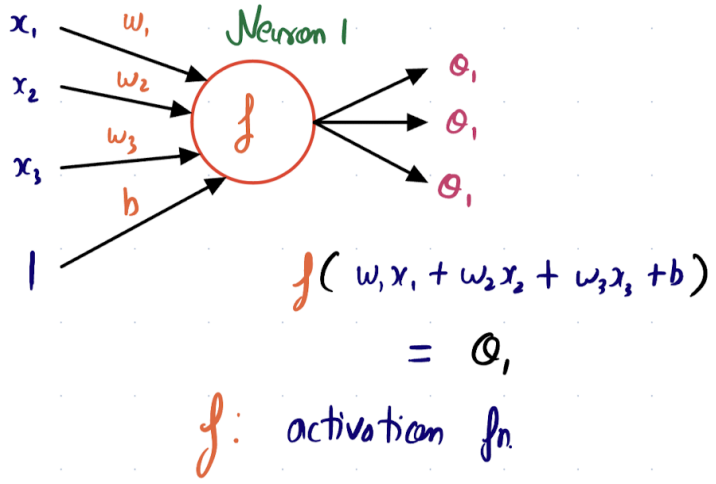
  Consider an artificial neuron.

  - It receives input features: x_1, x_2, x_3
  - Every input has a weight associated with it: w_1, w_2, w_3
    - These weights are multiplied by the input values, thereby telling us how important a given input is to the neuron.
  - Inputs are processed by taking the weighted sum.
  - Also, a bias term is added.
    - The net input becomes: $w_1x_1 + w_2x_2 + w_3x_3 + b = z$ (let)
  - There is a function, called activation function f, which is associated with a neuron
    - The neuron applies this function on the net input value: $f(z) = f(w_1x_1 + w_2x_2 + w_3x_3 + b)$
  - The result of this function becomes the output $o_1 = f(z)$
  - This output is then forwarded to other neurons.

  This flow of computations is known as **Forward Propagation.**

  Notice that we are going from left to right during Forward Propagation.

Artificial Neuron



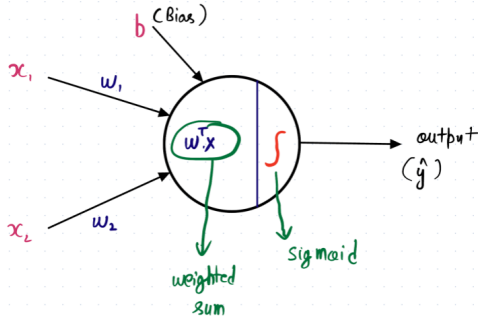
What happens if we put the sigmoid function as the activation function?

We get a Logistic Regression model, as the output of this becomes:

$o_1 =$ sigmoid($w_1x_1 + w_2x_2 + w_3x_3 + b$)

This neuron, where the activation is a sigmoid function is called a Logistic Regression Unit (LRU)

We can diagrammatically represent a neuron with 2 inputs as:



What if the model looked the same, but the activation function is different? Would it still be called the same NN?

If we replace the activation function to a hinge loss function, we get a Linear SVM model.

Forward propagation here would look like:-

- $z = w_1x_1 + w_2x_2 + \ldots + w_dx_d + b$
- $\hat{y} = f_{hinge}(z)$

Note:

- A single neuron is able to represent such powerful models, just imagine what will happen if we use multiple neurons.
- Those models would be able to represent really complex relations.

A brief history of Artificial Neural Networks

Perceptron

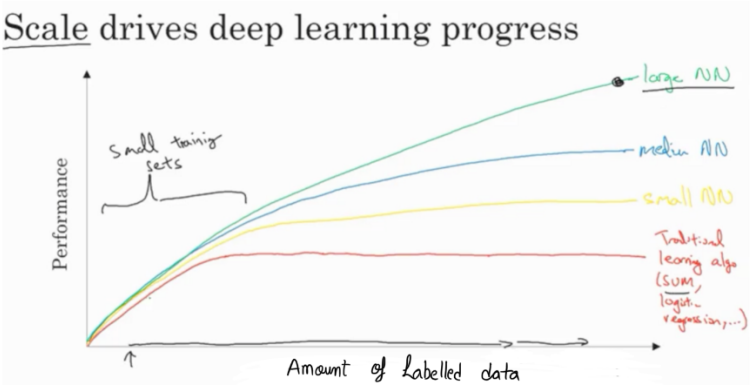
- Perceptron is the very first NN-based model. It was designed by Rosenblatt in 1957
- It is different from LRU, because it used a step function for activation, which is:

$$f_{perceptron}(x_i, w, b) = \begin{cases} 1, & \text{if } w^T x_i + b > 0. \\ 0, & \text{otherwise.} \end{cases}$$

Challenges:

- As we tried to increase the complexity of NN, they performed poorly.
- There was no optimal way of training the NN.
- This put the whole area in deep freeze for a decade or so.
- A breakthrough came in 1986 when **backpropagation** was introduced by Geoff Hinton and his team.
- This also brought up a lot of hype for artificial intelligence, But, all of that hype died down by 1990s
- There were two main bottlenecks:-
    - We didn't have computational power
    - Nor did we have enough data to train
    - Backprop was failing for NNs with more depth.
- This time period is called as **AI winter,** where the funding for AI dried up by 1995
- Meanwhile, Geoff Hinton continued his research and finally came up with a solution to this problem in 2006.
- Also, the discovery of using ReLu and Leaky ReLu as activation functions was another breakthrough.

How do NN fare against classical ML (based on training data)?



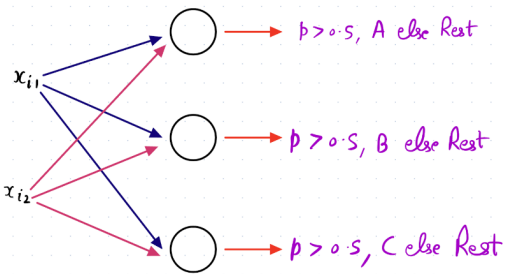
An LRU can only help in the case of binary classification (0 or 1).

How can we adapt a NN to do multi-class classification?

Suppose we wish to do multi-class classification for 3 classes: A, B, and C.

Recall multi-class classification:

- We calculated the probability that a given data point belongs to class A, B, or C respectively.
- Then, we returned the class with the highest probability as the answer.
- This gives us the intuition that perhaps, our output layer should have 3 outputs. One for each class.



We cannot perform multi-class classification using a sigmoid because we might get $p > 0.5$ for more than one class.

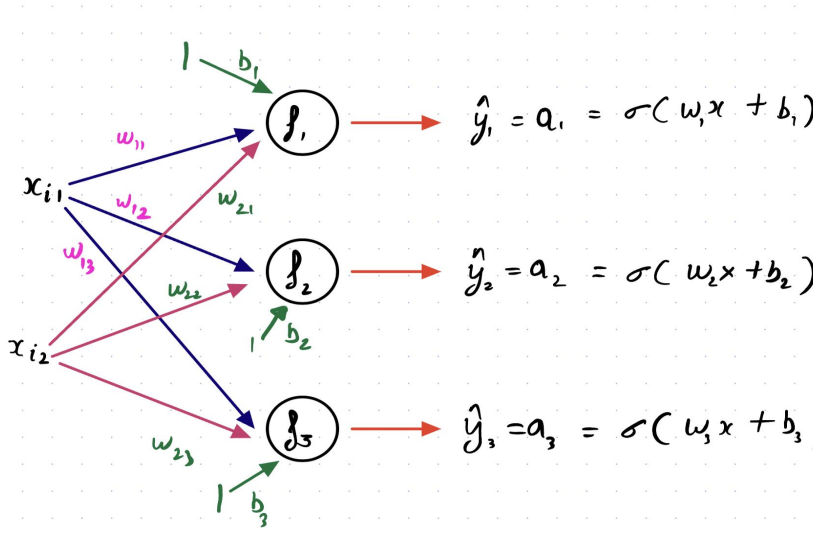
- Model will predict the presence of multiple classes in the output - [1, 1, 0], [1, 1, 1], [1, 0, 1]
- Conclusion: We want these probabilities values to sum to 1, as we had in Logistic Regression (p and 1-p).

It should be $p_A + p_B + p_C = 1$

In order to do this, we use the **softmax function** as activation in the neurons of the **output layer**.

$$p_i = \frac{e^{z_i}}{\sum_{i=0}^{k} e^{z_i}}$$

The model now looks like:-



Note:

- There will be 2*3 = 6 weights, and it'll be stored as a 2x3 matrix: W_{2x3}
- There will be 3 biases, one for each neuron, and it'll be stored as a 1x3 matrix: b_{1x3}

How will we calculate the loss for this multi-class classification NN Model?
We use Categorical Cross Entropy.

Cross Entropy (CE_i) for i^{th} datapoint will be:

$$CE_i = -\sum_{j=1}^{k} y_{ij} log(P_{ij})$$

where,
k -> number of classes
y_{ij} -> one hot encoded label. Ex: [1,0,0], [0,1,0], or [0,0,1]
P_{ij} -> Calculated Probability of datapoint belonging to class j
This can be seen as log loss extended to the multiclass setting. For k=2, we will get log loss formulation.

How to train NN?
Let, m -> no of training examples
d -> no of features
n -> no of classes/neurons in the output layer

Process to train a NN is:-

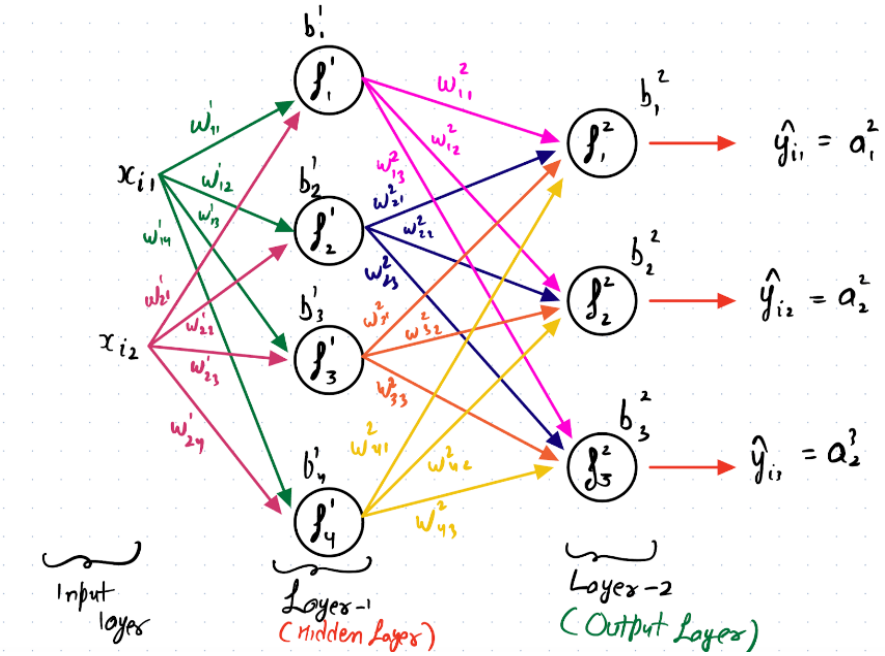
- Randomly Initialise parameters: W and b matrices
- Do forward propagation

- $Z = X_{m x d} . W_{d x n} + b_{1 x n}$
- $A = activation(Z)$

- Calculate the Loss
- Repeat until Loss converges
  - update $w_i = w_i - lr * (\partial Loss / \partial w_i)$
  - calculate the output using hypothesis and updated params
  - calculate the Loss

- **NN-3: Backpropagation and Activation functions**
  What are MLPs?
  If we wish to get a more complex decision boundary, we need to add another layer of neurons in the model. This is known as the **hidden layer**.
  - These models are known as MLPs (Multi Layer Perceptrons). Or N-layered NN
  - They are based on the idea of function composition.
  - We can have as many hidden layers as we want, however, the greater the number, the higher the risk of overfitting.
  - The activation of hidden layers also needs to always be **non-linear,** otherwise, we will not be able to get complex features.

  A 2-layer model looks like:-



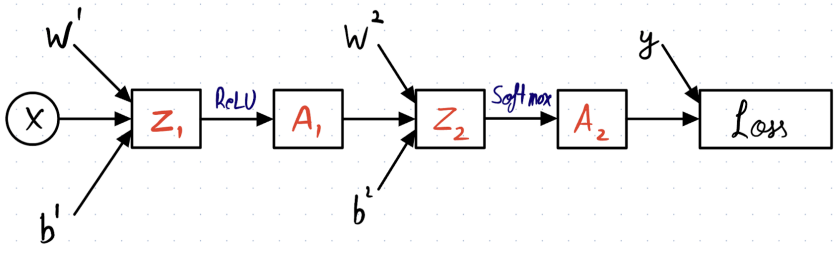
  Since there are multiple layers, we modify the notation to cater to it
  - Weights: $\mathbf{W^L_{ij}}$
  - Bias: $\mathbf{b^L_i}$

  Note:-

- Weights in layer-1 will be stored as a 2x4 matrix: $W^{1}_{2\,x\,4}$
- Biases in layer-1 will be stored as a 1x4 matrix: $b^{1}_{1\,x\,4}$
- Weights in layer-2 will be stored as a 4x3 matrix: $W^{2}_{4\,x\,3}$
- Biases in layer-2 will be stored as a 1x3 matrix: $b^{2}_{1\,x\,3}$

Why do we need to add a hidden layer to increase complexity?
The idea is that Stacking a non-linearity over a linear function, and repeating the process helps create complex features

What does forward propagation look like for this NN?
Let h -> no of neurons in the hidden layer.
m -> no of training examples
d -> no of features
n -> no of classes/neurons in the output layer

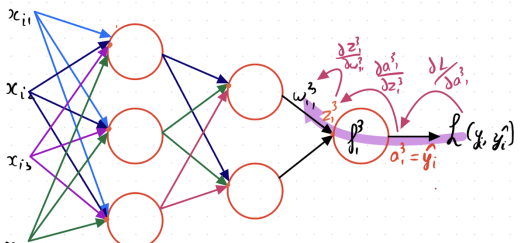


Forward propagation:-

$$Z^{1}_{m\,x\,h} = X_{m\,x\,d}.W^{1}_{d\,x\,h} + b^{1}_{1\,x\,h}$$
$$A^{1}_{m\,x\,h} = f^{1}(Z^{1}_{m\,x\,h})$$
$$Z^{2}_{m\,x\,n} = A^{1}_{m\,x\,h}.W^{2}_{h\,x\,n} + b^{2}_{1\,x\,n}$$
$$A^{2}_{m\,x\,n} = f^{2}(Z^{2}_{m\,x\,n})$$

How do we train this complex NN? What is Backpropagation?

- In order to update the parameters, we need to find their gradients. For this, we use the Backpropagation algorithm, where we traverse from right to left in the NN.
- It is based on the concept of chain rule of differentiation.



Gradient of w^3_{11}

We'll encounter a^3_1 while going from loss towards w^3_{11}

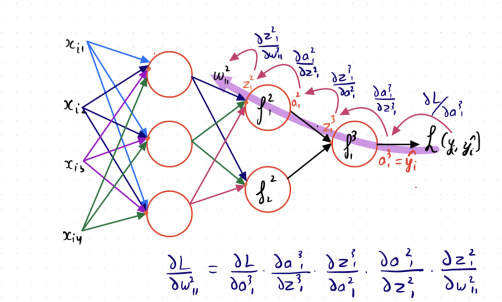
Therefore gradient: $\frac{\partial L}{\partial w^3_{11}} = \frac{\partial L}{\partial a^3_1} . \frac{\partial a^3_1}{\partial z^3_1} . \frac{\partial z^3_1}{\partial w^3_{11}}$

Gradient of w^2_{11}

We'll encounter a^3_1 and a^2_1 while going from loss towards w^2_{11}

Therefore gradient:

$$\frac{\partial L}{\partial w^3_{11}} = \frac{\partial L}{\partial a^3_1} . \frac{\partial a^3_1}{\partial z^3_1} . \frac{\partial z^3_1}{\partial a^2_1} . \frac{\partial a^2_1}{\partial z^2_1} . \frac{\partial z^2_1}{\partial w^2_{11}}$$



Gradient of w^1_{11}

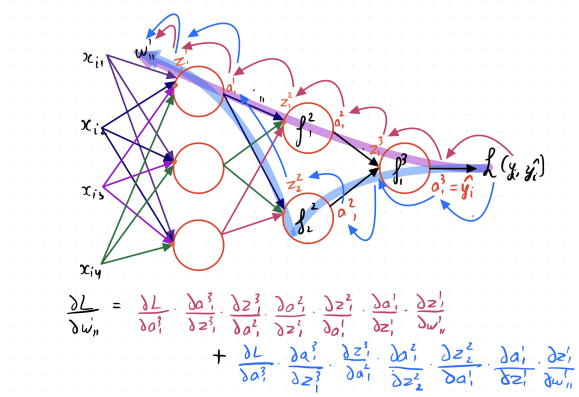
There are 2 possible paths to reach w^1_{11}

Path-1 : L -> a^3_1 -> a^2_1 -> a^1_1 -> w^1_{11}

Path-2 : L -> a^3_1 -> a^2_2 -> a^1_1 -> w^1_{11}

We need to combine the derivatives from path 1 and 2 by adding them up.

Therefore gradient:



$$\frac{\partial L}{\partial w^1_{11}} = \frac{\partial L}{\partial a^3_1} . \frac{\partial a^3_1}{\partial z^3_1} . \frac{\partial z^3_1}{\partial a^2_1} . \frac{\partial a^2_1}{\partial z^2_1} . \frac{\partial z^2_1}{\partial a^1_1} . \frac{\partial a^1_1}{\partial z^1_1} . \frac{\partial z^1_1}{\partial w^1_{11}} + \frac{\partial L}{\partial a^3_1} . \frac{\partial a^3_1}{\partial z^3_2} . \frac{\partial z^3_2}{\partial a^2_1} . \frac{\partial a^2_2}{\partial z^2_2} . \frac{\partial z^2_2}{\partial a^1_1} . \frac{\partial a^1_1}{\partial z^1_1} . \frac{\partial z^1_1}{\partial w^1_{11}}$$

What are the different activation functions?

- Sigmoid function
- Hyperbolic tan function: $\frac{e^z - e^{-z}}{e^z + e^{-z}}$
- ReLu: $ReLu(z) = max(z, 0)$
- Leaky ReLu: $Leaky\ ReLu(z) = max(z, \alpha z)$; α is a small gradient that we add

What is the vanishing gradient problem?

- Downside of both sigmoid and tanh is that their gradient is ~0, for most of the values of z
- This hampers the gradient descent process, as the calculated gradients become very small.
- For eg Suppose we wish to update weight w^1_{11}. its gradient is calculated as:

$$\frac{\partial L}{\partial w^1_{11}} = \frac{\partial L}{\partial a^3_1}\left[\frac{\partial a^3_1}{\partial a^2_{11}} \cdot \frac{\partial a^2_{11}}{\partial a^1_{11}} \cdot \frac{\partial a^1_{11}}{\partial w^1_{11}} + \frac{\partial a^3_1}{\partial a^2_{21}} \cdot \frac{\partial a^2_{21}}{\partial a^1_{12}} \cdot \frac{\partial a^1_{12}}{\partial w^1_{11}}\right]$$

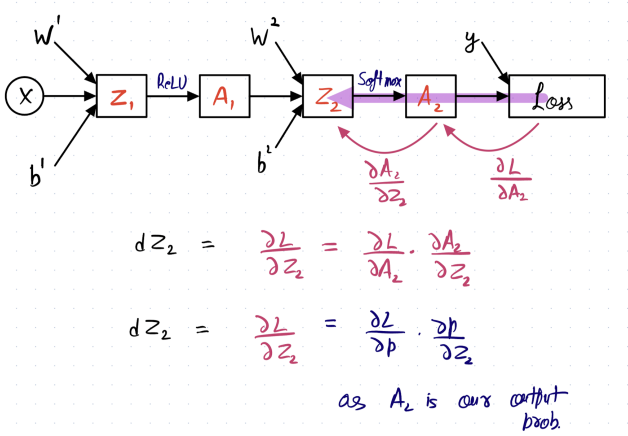
- So, the product of these terms inside the bracket will become very small.
- In fact, as the number of layers in the NN increase, this product will become smaller and smaller.

Backprop for MLP

- **Calculating dZ^2**

$dZ^2 = \frac{\partial L}{\partial Z^2} = \frac{\partial L}{\partial A^2} \cdot \frac{\partial A^2}{\partial Z^2}$

$dZ^2 = \frac{\partial L}{\partial p} \cdot \frac{\partial p}{\partial Z^2} = p_i - I(i == k)$

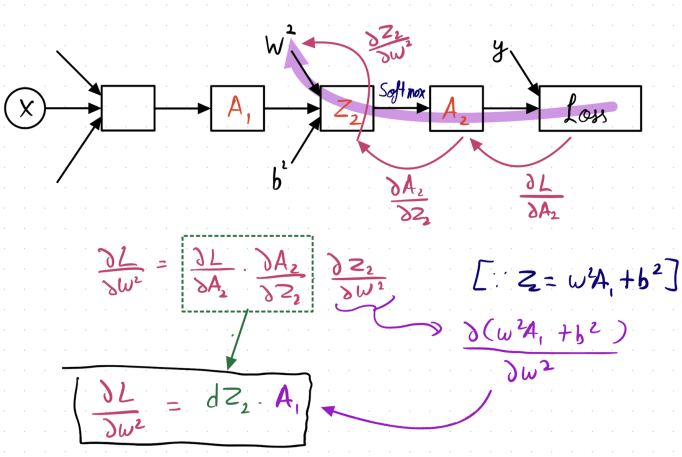


- **Calculating dW^2**

$dW^2 = \frac{\partial L}{\partial W^2} = \frac{\partial L}{\partial A^2} \cdot \frac{\partial A^2}{\partial Z^2} \cdot \frac{\partial Z^2}{\partial W^2}$

$dW^2 = dZ^2 \cdot \frac{\partial Z^2}{\partial W^2}$

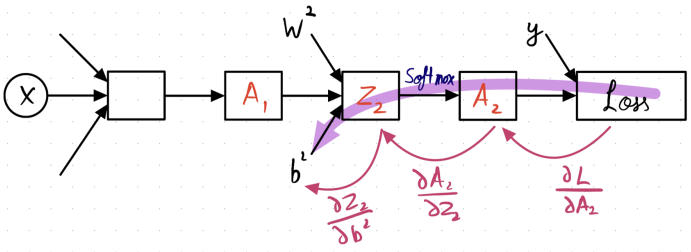
$dW^2 = dZ^2 \cdot A^1$



- **Calculating db^2**

$$db^2 = \frac{\partial L}{\partial b^2} = \frac{\partial L}{\partial A^2} \cdot \frac{\partial A^2}{\partial Z^2} \frac{\partial Z^2}{\partial b^2}$$

$$db^2 = dZ^2 \cdot \frac{\partial Z^2}{\partial b^2} = dZ^2$$

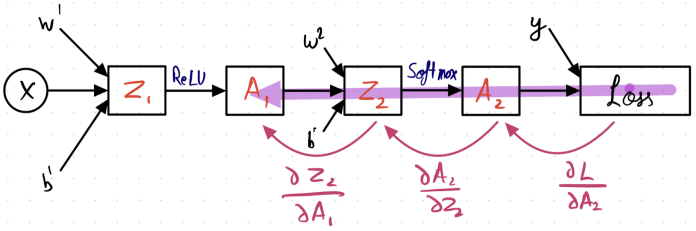


$$\frac{\partial z_2}{\partial b^2} = \frac{\partial (w^2 A_1 + b^2)}{\partial b^2} = 1$$

- **Calculating dA^1**

$$dA^1 = \frac{\partial L}{\partial A^1} = \frac{\partial L}{\partial A^2} \cdot \frac{\partial A^2}{\partial Z^2} \frac{\partial Z^2}{\partial A^1}$$

$$dA^1 = dZ^2 \cdot \frac{\partial Z^2}{\partial A^1} = dZ^2 \cdot W^2$$



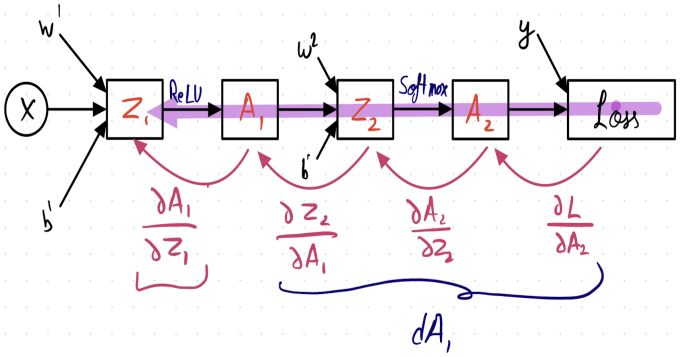
$$\frac{\partial z_2}{\partial A_1} = \frac{\partial (w^2 A_1 + b')}{\partial A_1}$$

$$\boxed{\frac{\partial z_2}{\partial A_1} = w^2}$$

- **Calculating dZ^1**

$$dZ^1 = \frac{\partial L}{\partial Z^1} = \frac{\partial L}{\partial A^2} \cdot \frac{\partial A^2}{\partial Z^2} \frac{\partial Z^2}{\partial A^1} \frac{\partial A^1}{\partial Z^1}$$

$$dZ^1 = dA^1 \cdot \frac{\partial A^1}{\partial Z^1}$$



$$dZ^1 = dA^1 \cdot \frac{\partial A^1}{\partial Z^1}$$

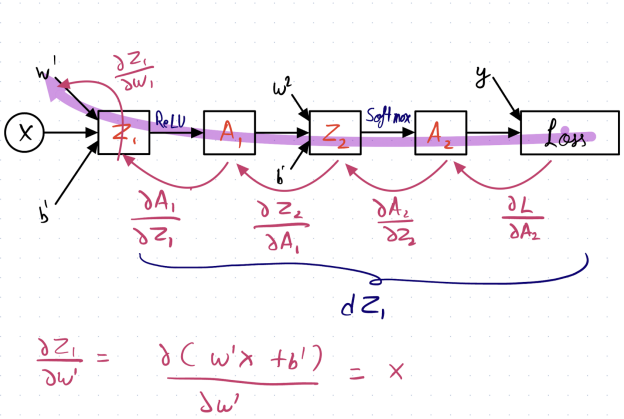
$$\frac{\partial L}{\partial Z_1} = \partial A_1 \cdot \frac{\partial A_1}{\partial Z_1}$$ → Can be 0 or 1

$$\left(\partial A_1 * 0 \text{ if } Z_1 \leq 0 \right)$$

- **Calculating dW^1**

$$dW^1 = \frac{\partial L}{\partial W^1} = \frac{\partial L}{\partial A^2} \cdot \frac{\partial A^2}{\partial Z^2} \frac{\partial Z^2}{\partial A^1} \frac{\partial A^1}{\partial Z^1} \frac{\partial Z^1}{\partial W^1}$$

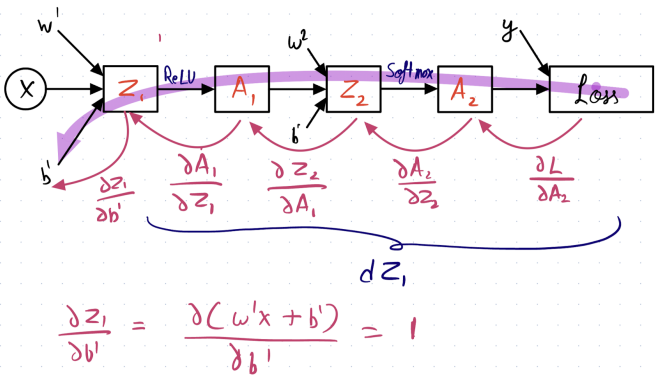
$$dW^1 = dZ^1 \frac{\partial Z^1}{\partial W^1} = dZ^1.X$$



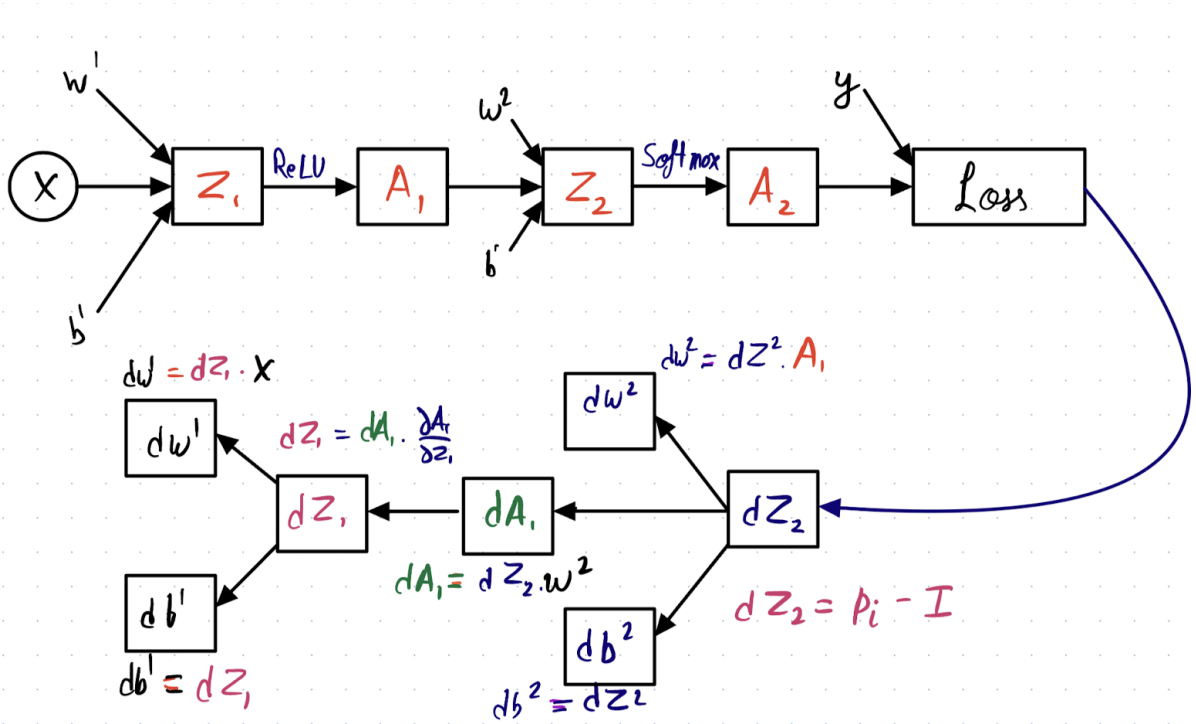
- **Calculating db^1**

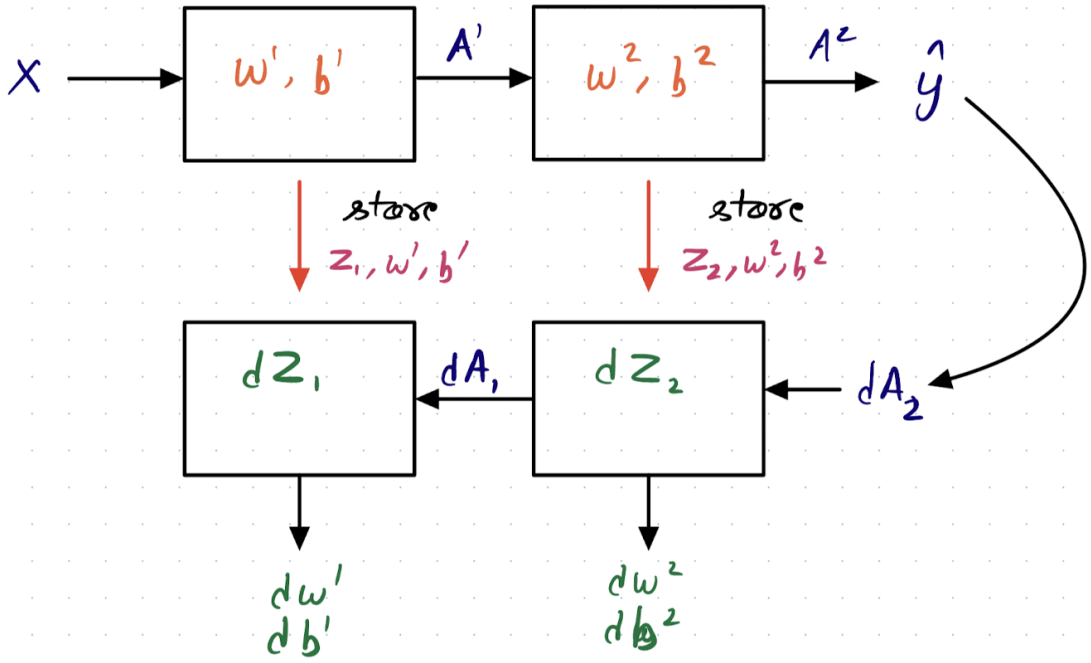
$$db^1 = \frac{\partial L}{\partial b^1} = \frac{\partial L}{\partial A^2} \cdot \frac{\partial A^2}{\partial Z^2} \frac{\partial Z^2}{\partial A^1} \frac{\partial A^1}{\partial Z^1} \frac{\partial Z^1}{\partial b^1}$$

$$db^1 = dZ^1 . \frac{\partial Z^1}{\partial b^1} = dZ^1.1 = dZ^1$$



Summarizing forward and backward prop for MLP





While performing forward prop,

- we store/cache the value of Zj, Wj, bj in order to use them during back prop

Can we use Neural Networks for the Regression task?

- Yes, if the activation function for the output layer is a linear function, then NN will do regression.
- The activations for intermediate layers still need to be non-linear, otherwise, NN will not be able to map complex relationships.

Tensorflow and keras

Tensorflow

```
[ ]  import tensorflow as tf
     tf.__version__
```

```
'2.9.2'
```

TensorFlow is the premier open-source deep learning framework developed and maintained by Google

Keras

- Using TensorFlow directly can be challenging,
- In TensorFlow 2, Keras has become the default high-level API
- No need to separately install keras
- The modern tf.keras API brings Keras's simplicity and ease of use to the TensorFlow project.

Why keras ?

- easy to build and use giant deep learning models

- **Light-weight and quick**
- can support other backends as well besides tensorflow, eg: Theano
- Open source

dir()

It returns list of the attributes and methods of any objec

- dir(tf.keras)
- dir(tf.keras.activation)
- dir(tf.data)

Ways of writing code in Keras

- Sequential API
- Functional API

Keras Sequential API

- Simplest and recommended API to start with
- Called as “Sequential” because we add layers to the model one by one in a
- linear manner, from input to output.
- You can select optimizers, loss functions and metrics while writing code

How to import ?

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
```

Dense layer helps us define one layer of a Feedforward NN.

Example model using Sequential API

```
[ ] model = Sequential([
                        Dense(64, activation="relu"), #hidden dense layer with 64 neuron units
                        Dense(4, activation="softmax") #output layer with 4 units and softmax activation
    ])
```

Note : The layers in the sequential model interact with each other therefore we don't need to define the input shape for all the layers.

We can check model weights using model.weights

model.add()

- Instead of passing the list of layers as an argument while creating a model instance, we can use the add method.

```
model = Sequential()
model.add(Dense(64, activation="relu", input_shape=(11,)))
model.add(Dense(4, activation="softmax"))
```

model.summary()

To print the summary of model we have created

```
[ ] model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_4 (Dense)             (None, 64)                768

 dense_5 (Dense)             (None, 4)                 260

=================================================================
Total params: 1,028
Trainable params: 1,028
Non-trainable params: 0
_________________________________________________________________
```

Custom names

- keras has provided the names by itself.
- We can also give custom names to the layer as well

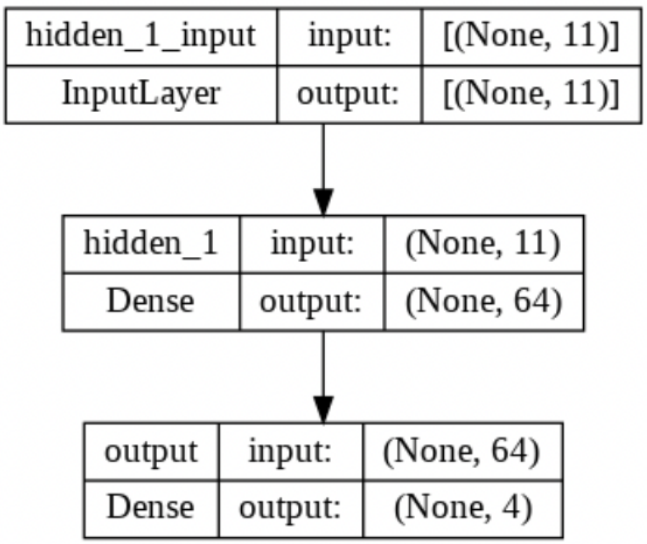
```
model = Sequential([
                    Dense(64, activation="relu", input_shape=(11,), name="hidden_1"),
                    Dense(4, activation="softmax", name="output")
])
```

```
[ ] model.summary()
```

```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 hidden_1 (Dense)            (None, 64)                768

 output (Dense)              (None, 4)                 260

=================================================================
Total params: 1,028
Trainable params: 1,028
Non-trainable params: 0
_________________________________________________________________
```

Plotting model

```
from tensorflow.keras.utils import plot_model

plot_model(model,
    to_file='model.png',
    show_shapes=True, show_layer_names=True)
```



Weights and Bias Initializer

In keras, in Dense layer,

1) the biases are set to zero (zeros) by default
2) the weights are set according to glorot_uniform

For own custom initializer, we can use **bias_initialiser** and **kernel_initialiser**

Compiler - loss and optimizer

What things to decide while compiling model ?

- Loss
- Optimizer

we can define a list of metrics which we might want to track during the training, like accuracy

```
[ ]  model_2C = Sequential([
                Dense(64, activation="relu", input_shape=(11,)),
                Dense(1, activation="sigmoid")])

     # new piece of code
     model_2C.compile(
         optimizer = "adam", # stochastic gradient descent, adam, rmsprop, adadelts
         loss = "binary_crossentropy", # sigmoid loss, # mean_squared_error, categorical_crossentropy,
                                       #sparse_categorical_crossentropy, binary_crossentropy
         metrics = ["accuracy"]
     )
```

- Another way to define **optimizer = keras.optimizers.Adam(learning_rate=0.01)**
- We can also pass customized loss and optimizer functions in keras models.
- These metrics will be calculated and saved after each epoch (one pass of whole data to update the model).

Epoch

- To avoid memory issues, data is passed in small batches instead of whole
- Each pass of mini-batch is called an iteration.
- Each pass of whole datasets is called an Epoch.
- One epoch means that each sample in the training dataset has had an opportunity to update the internal model parameters.

Training : model.fit ()

It means updating the weights using the optimizer and loss functions on the dataset.

```
model.fit(X_train, y_train)

X_train = (num_samples, num_features)

y_train = (num_samples, num_classes) or y_train = (num_samples, )
```

Arguments we can pass to this method

- epochs (number of epochs you want to train for)
- batch_size (Batch size usually in form of 2^x like 4,8,16,32)
- Validation_split (size of validation data)
- verbose (0 for silent training, 1 to print each iteration)

```
%%time
model.fit(X_train, y_train, epochs=10, batch_size=256, validation_split=0.1, verbose=1)

## no of iterations: ( 10847 (training size) - 1084.7 (validation split) )/(256) == 39
```

```
Epoch 1/10
39/39 [==============================] - 1s 8ms/step - loss: 1.3526 - accuracy: 0.3514 - val_loss: 1.2917 - val_accuracy: 0.4184
Epoch 2/10
39/39 [==============================] - 0s 3ms/step - loss: 1.2592 - accuracy: 0.4327 - val_loss: 1.2054 - val_accuracy: 0.4525
Epoch 3/10
39/39 [==============================] - 0s 3ms/step - loss: 1.1722 - accuracy: 0.4666 - val_loss: 1.1224 - val_accuracy: 0.4876
Epoch 4/10
39/39 [==============================] - 0s 3ms/step - loss: 1.0951 - accuracy: 0.4942 - val_loss: 1.0554 - val_accuracy: 0.5051
Epoch 5/10
39/39 [==============================] - 0s 3ms/step - loss: 1.0361 - accuracy: 0.5233 - val_loss: 1.0032 - val_accuracy: 0.5318
```

Note: After training, weights now will follow normal distribution and biases will not be Zero now

History object

- model.fit returns a history object which contains the record of progress NN training.
- History object contains records of loss and metrics values for each epoch.
- It's an alternative to dir(). __dict__ attribute can be used to retrieve all the keys associated with the object on which it is called.

```
history = model.fit(X_train, y_train, validation_data = (X_val, y_val),  epochs=500, batch_size=512, verbose=0)
history.__dict__.keys()
```

```
dict_keys(['validation_data', 'model', '_chief_worker_only', '_supports_tf_logs', 'history', 'params', 'epoch'])
```

history object's dictionary has another dictionary with key "history" inside it

```
history.history.keys()
```

```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

Model has saved all the loss and metrics values for each epoch inside the history dictionary where all the values are stored in different lists.

```
epochs = history.epoch
loss = history.history["loss"]
accuracy = history.history["accuracy"]
val_loss = history.history["val_loss"]
val_accuracy = history.history["val_accuracy"]
```

Prediction and Evaluation

Evaluate the model

loss, accuracy = model.evaluate(X_test, y_test)

- model.evaluate **returns the loss value & metrics value** for the model.
- **weights/parameters are not updated during evaluation** (and prediction) which means **only forward pass, no backward pass**

```
[ ] loss, accuracy = model.evaluate(X_test, y_test)
    print('Test Set')
    print("Loss value : ", loss)
    print("Accuracy   : ", accuracy)
```

```
42/42 [==============================] - 0s 2ms/step - loss: 0.5842 - accuracy: 0.7634
Test Set
Loss value :  0.5841851830482483
Accuracy   :  0.7634328603744507
```

Predictions

pred = model.predict(X_test)

```
[ ]  pred = model.predict(X_test)
     pred

     42/42 [==============================] - 0s 1ms/step
     array([[9.99999940e-01, 2.10884883e-17, 1.79274864e-33, 0.00000000e+00],
            [1.13163900e-03, 1.37660122e-02, 1.01408757e-01, 8.83693516e-01],
            [1.93726644e-02, 2.39155009e-01, 2.92279452e-01, 4.49192822e-01],
            ...,
            [1.64477840e-01, 8.35509479e-01, 1.26833165e-05, 9.64229950e-15],
            [6.74094200e-01, 3.25786144e-01, 1.19630786e-04, 1.66241088e-12],
            [1.01807564e-02, 2.50488132e-01, 7.39329159e-01, 2.02724095e-06]],
           dtype=float32)
```

- **To get predictions on unseen data, model.predict method is used**
- **It returns raw output from the model (i.e. probabilities of an observation belong to each one of the 4 classes**
- sum of probabilities of an observation belong to each of the 4 classes will be 1 i.e, **np.sum(pred, axis=1)**

To know the class an observation belongs to, using these 4 probability values

- Find the index having the largest probability and that will be the predicted class.
- **pred_class = np.argmax(pred, axis = 1)**

To check accuracy of the model using sklearn's accuracy_score

```
from sklearn.metrics import accuracy_score
acc_score = accuracy_score(y_test, pred_class)
```

Callbacks

A callback **defines a set of functions** which are **executed at different stages of the training** procedure.

They can be **used to view internal states of the model** during training.

- For example, we may **want to print loss, accuracy or lr every 2000th epoch**.

Examples:

1. **on_epoch_begin** : function will execute before every epoch
2. **on_epoch_end** : function will execute after every epoch
3. **verbose**=1, model training prints associated data after every epoch
4. **verbose**=0, model training prints nothing

Customized callback example

```
class VerboseCallback(tf.keras.callbacks.Callback):
    # runs only before the training starts
    def on_train_begin(self, logs=None):
          print("Starting training...")

    # runs after every epoch
    def on_epoch_end(self, epoch, logs = None):
      if epoch % 50 == 0:
         print(f'Epoch {str(epoch).zfill(3)}', '- loss : ', logs['loss'], '- Acc : ', logs['accuracy'])

    # runs once training is finished
    def on_train_end(self, logs=None):
          print("...Finished training")
```

- The custom class will inherit from tf.keras.callbacks.Callback.
- All methods in keras.callbacks.callback class will be available for our customized class, and we can also override them.

We will have to **pass a list of callback objects** to callbacks argument of the fit method

```
history = model.fit(X_train, y_train, epochs=500, batch_size=256,
                    validation_split=0.1, verbose=0, callbacks=[VerboseCallback()])
```

```
Starting training...
Epoch 000 - loss :  0.5516670346260071 - Acc :  0.7737144231796265
Epoch 050 - loss :  0.5456981658935547 - Acc :  0.7760704755783081
Epoch 100 - loss :  0.5408483743667603 - Acc :  0.776172935962677
Epoch 150 - loss :  0.5364638566970825 - Acc :  0.7808850407600403
Epoch 200 - loss :  0.5334064364433289 - Acc :  0.7805777788162231
Epoch 250 - loss :  0.530765175819397 - Acc :  0.7842655181884766
Epoch 300 - loss :  0.5280439853668213 - Acc :  0.7856996655464172
Epoch 350 - loss :  0.524202823638916 - Acc :  0.7886703610420227
Epoch 400 - loss :  0.5229038000106812 - Acc :  0.7881581783294678
Epoch 450 - loss :  0.5198482275009155 - Acc :  0.7865191698074341
...Finished training
```

Note: We **can pass callback objects to evaluate and predict** method as well.

The parent class tf.keras.callbacks.Callback supports various kinds of methods which we can override

- Global methods
  at the beginning/ending of training
- Batch-level method
  at the beginning/ending of a batch
- Epoch-level method
  at the beginning/ending of an epoch

Other examples include:

1. CSVLogger - save history object in a csv file csv_logger = keras.callbacks.CSVLogger("file_name.csv")
2. EarlyStopping - stop the training when model starts to overfit
3. ModelCheckpoint - saves the intermediate model weights
4. LearningRateScheduler - control/change Learning Rate in between epoch

Tensorboard

- It is used to **closely monitor the training process**
- It can be used to visualize information regarding the training process like
  - ❖ Metrics - loss, accuracy
  - ❖ Visualize the model graphs
  - ❖ Histograms of W, b, or other tensors as they change during training - distributions
  - ❖ Displaying images, text, and audio data

Ways to install

pip install tensorboard

conda install -c conda-forge tensorboard

To load tensorboard in the notebook

%load_ext tensorboard

TensorBoard will **store all the logs in this log directory**.

```
log_folder = 'logs'
```

- It will read from these logs in order to display the various visualizations.

If we want to reload the TensorBoard extension, we can use the reload magic method

```
%reload_ext tensorboard
```

Import

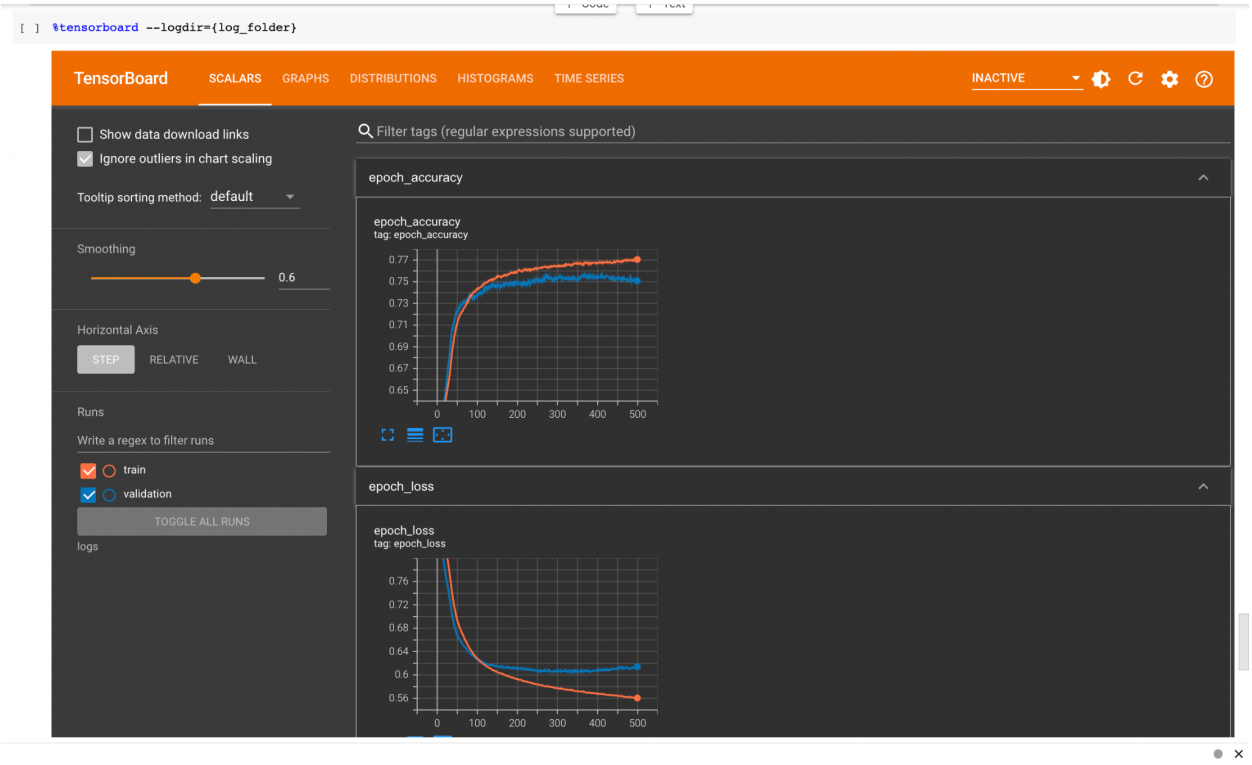
```
model = create_model()
from tensorflow.keras.callbacks import TensorBoard
tb_callback = TensorBoard(log_dir=log_folder, histogram_freq=1)
history = model.fit(X_train, y_train, epochs=500, batch_size=512,
                    validation_data = (X_val, y_val), verbose=0, callbacks=[tb_callback])
```

Callback arguments:

- **log_dir** - (Path)
  - directory where logs will be saved
  - This directory should not be used by any other callbacks.
- **update_freq** - (int/str)
  - how frequently losses/metrics are updated.
  - when set to batch, losses/ metrics will be updated after every batch/iteration
  - when set to an integer N, updation will be done after N batches
  - when set to 'epoch', updation will be done after every epoch
- **histogram_freq** - (int)
  - how frequently (in epochs) histograms(Distribution of W) will be updated.
  - Setting this to 0 means, histograms will not be computed.
- **write_graph** - (Bool), True if we want to visualize our training process
- **write_images** - (Bool), True if we want to visualize our model weights

Launch tensorboard using following command

```
%tensorboard --logdir={log_folder}
```



Weight Initialization

Why need different Optimizer techniques ?

Ans: When training Deep NN, the model tends to have immense training time due to

- Large number of layers in NN
- Large number of neurons in NN

This makes the parameter (weight matrix) of the NN to be large in size.

What are the issues with Deep NN ?

1. Dead Neuron: When the weight $W = 0$ and bias $b = 0$ for all the layers of the NN

- As activation function is Relu which means :

$$\frac{\partial relu(z)}{\partial x} = 1 \text{ if } Z > 0 \text{ and } \frac{\partial relu(z)}{\partial x} = 0 \text{ if } Z \leq 0$$

- Thus making weight updation to be zero, meaning the NN doesn't learn during training time
- Similarly, when $W = k$ and bias $b = k$ for all the layers of the NN, where k is a constant
- The model acts as a single neuron NN, inspite of there being N number of Neurons.

2. Exploding Gradients

- If the Deep NN uses linear activation function for all of its L layers

$$a = g(z) = z$$

- Then the Weight Matrix = [W^1, W^2, W^3,..... W^{L-1}, W^L], and the final layer output is:

$$\hat{y} = g(W^L \times a^{L-1})$$

As Activation being linear: $g(W^L a^{L-1}) = W^L a^{L-1}$ and $a^{L-1} = g(W^{L-1} a^{L-2})$:

$$\hat{y} = W^L \times g(W^{L-1} \times a^{L-2})$$

Since the NN has layers $\in [1, L]$:

$$\hat{y} = W^L W^{L-1} ..... \times W^2 W^1 X$$

- Now for a Deep NN, L is a large value, which makes $\prod_{i=1}^{L} W^i$ will be a very large value
- Therefore the gradient values becomes exponentially high

How to avoid these issues ?

Weight Initialization strategies

1. Uniform Distribution: We initialize the weights as:

$$w^k_{ij} \sim uniform\ [\frac{-1}{\sqrt{fan_{in}}}, \frac{1}{\sqrt{fan_{out}}}\ ]$$

- Where fan_{in} is the number of input to a neuron while fan_{out} is the number of output of the neuron

2. Glorot/Xavier init:
   a. Normal Distribution

$$w^k_{ij} \sim N(0, \sigma_{ij}),\ where\ \sigma_{ij} = \sqrt{\frac{2}{fan_{in} + fan_{out}}}]$$

   b. Uniform Distribution

$$w^k_{ij} \sim uniform\ [\frac{-\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}}, \frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}}\ ]$$

   Note: Used when activation function is tanh

3. He Init:
   a. Normal Distribution

$$w^{k}_{ij} \sim N(0, \sigma_{ij}),\ where\ \sigma_{ij} = \sqrt{\frac{2}{fan_{in}}}]$$

b. Uniform Distribution

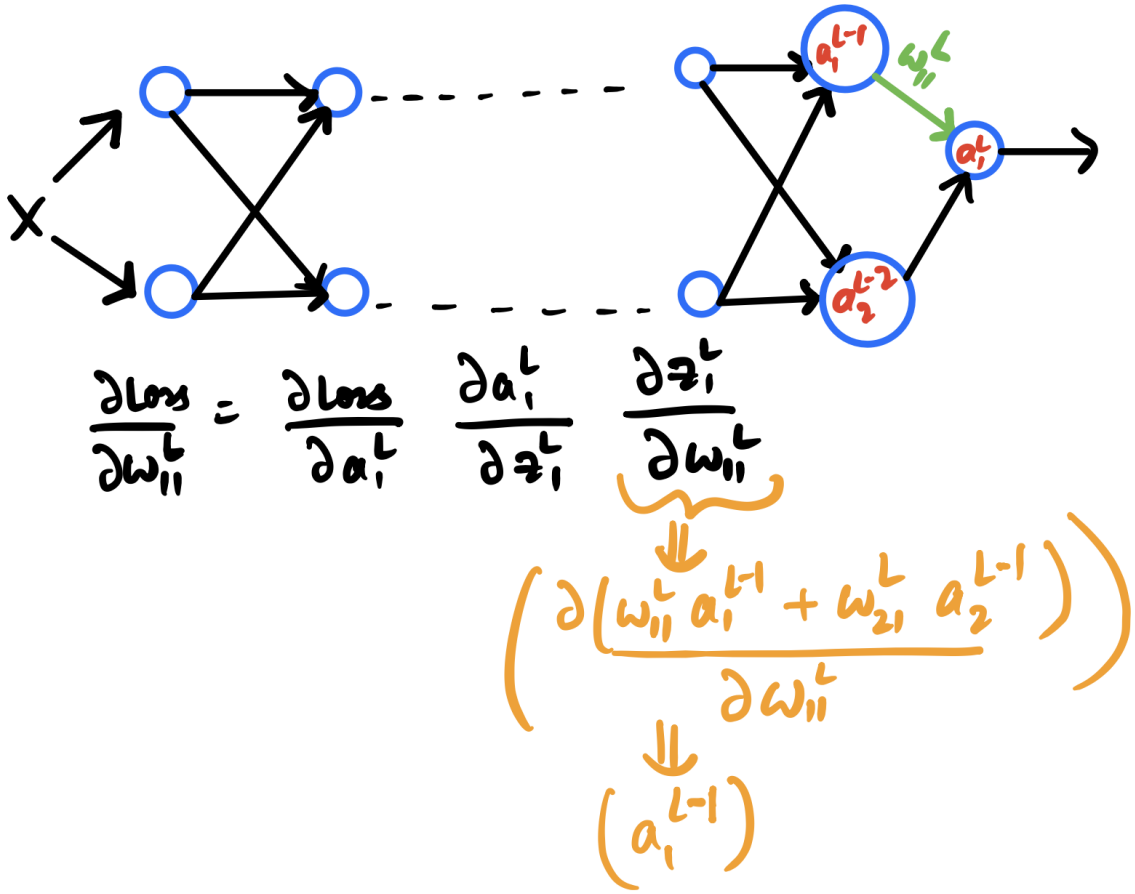
$$w^{k}_{ij} \sim uniform\ [\frac{-\sqrt{6}}{\sqrt{fan_{in}}}, \frac{\sqrt{6}}{\sqrt{fan_{in}}}\ ]$$

Note: Used when activation function is ReLU

Why need to initialize the weight based on input and output of the neuron ?

derivative of Z^{L}_{1} w.r.t w^{L}_{11} is defined as:

$$\frac{\partial Z^{L}_{1}}{\partial w^{L}_{11}} = a^{L-1}_{1} = activation(Z^{L-1}_{1})$$



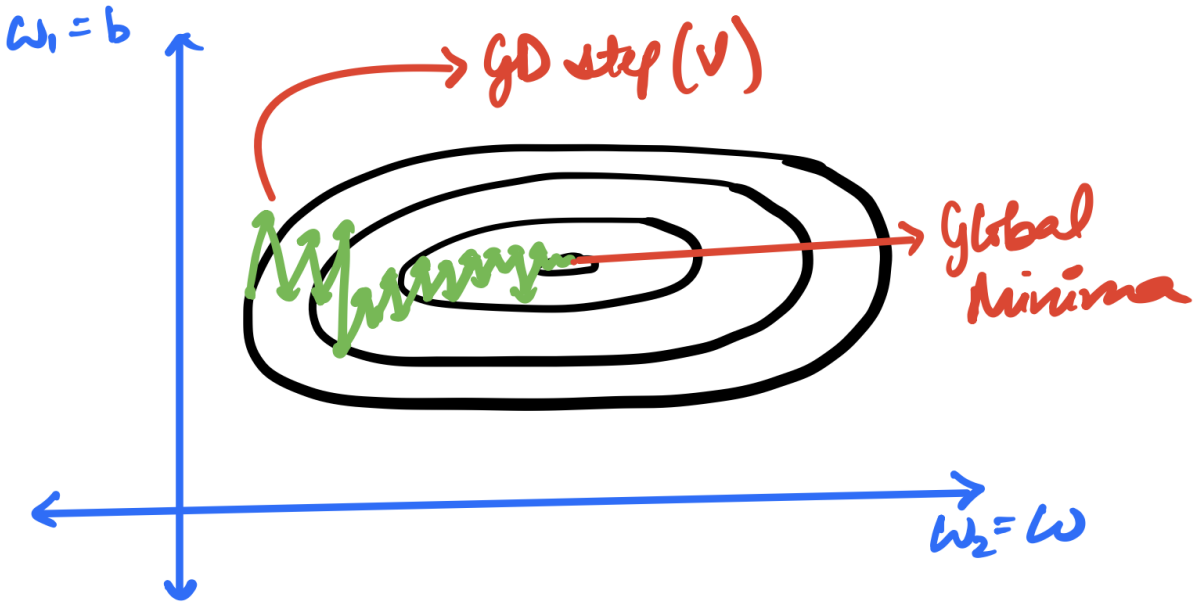
And observe Z^{L-1}_{1} is nothing but a function of weights = [W^{1}, W^{2},..... W^{L-1}]

- Hence if $Z^{L-1}{}_1$ has greater number of inputs, the value for each weight value is influenced drastically leading to exploding gradient

Optimizer

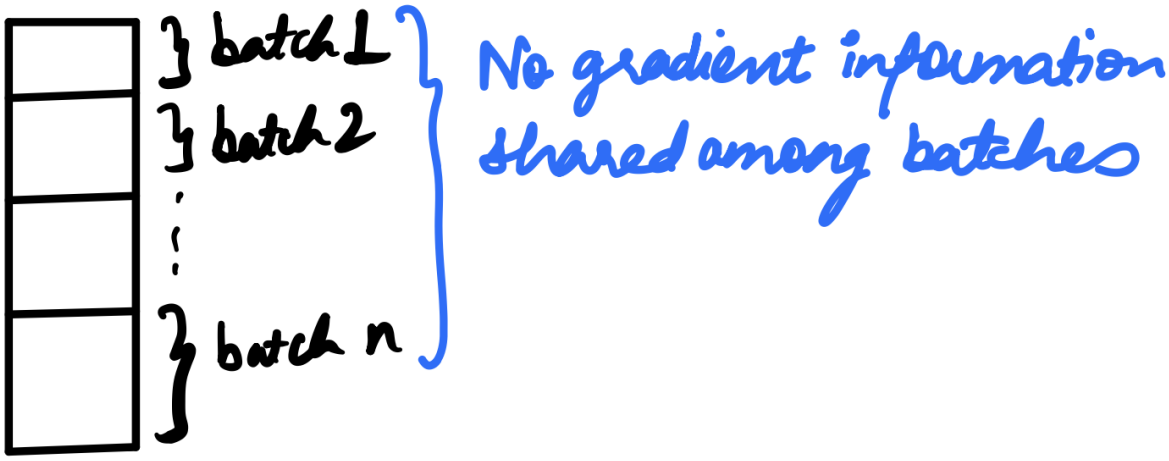
Why does SGD and Mini Batch Gradient Descent (GD), take so many epochs while training Deep NN ?

Ans: Mini-batch GD takes steps (V) where the GD tends to:
- Move in direction where it will never reach minima
- Hence due to all these noisy steps, the GD takes so many epochs



Why does mini-Batch GD have noisy steps ?

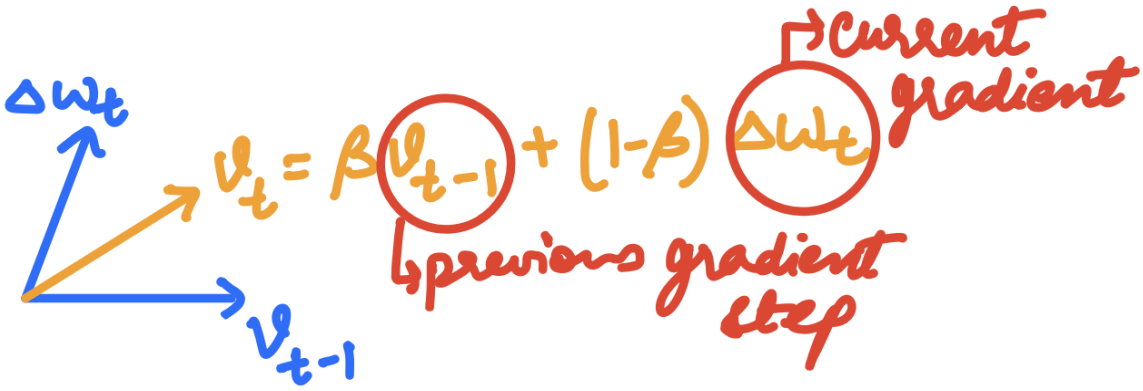
Ans: Because, training data is divided into batches
- And for some batch the model has very small loss
- while for a few batch, the loss is quite high
- Making the gradients of weights fluctuate



How to reduce these noisy steps ?

Ans: By taking some weighted average (β) from the previous Optimizer Step (V_{t-1}) along with the current gradient (Δw) , hence:

$$V_t = \beta V_{t-1} + (1 - \beta)\Delta w_t$$



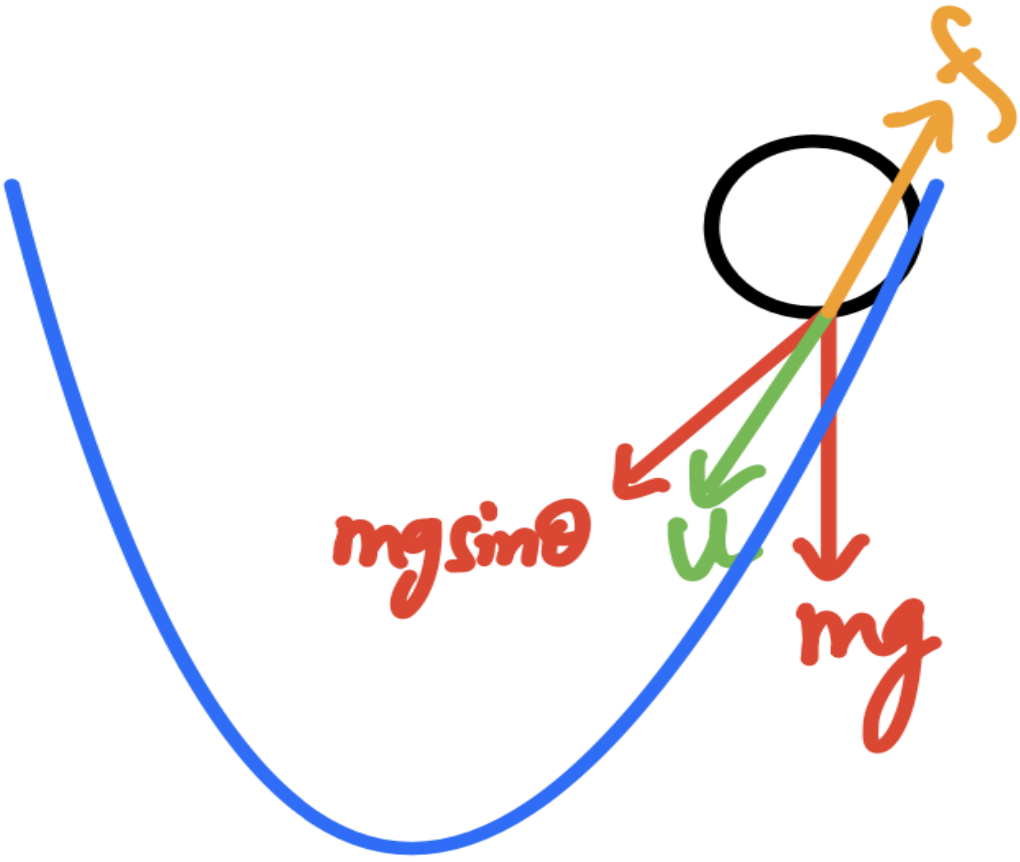
Note: t denotes t^{th} iteration, where 1 iteration = Forward + Backward Propagation

With the weightage (β) being introduced,

- The direction of V_3 will tend to be influenced by all previous gradients ΔW_1, ΔW_2 along with the current gradient ΔW_3
- Thus making the optimizer to take a step in the direction such that it avoids the noisy step
- This is known as Exponential Moving Average

This idea of Exponential Moving Average can be considered as a ball moving down a hill thus:

- $\beta \rightarrow$ Friction
- $V_{t-1} \rightarrow$ Velocity/Momentum
- $\Delta w_t \rightarrow$ Acceleration



How does Gradient Descent implement Exponential Moving Average ?

Ans: for some iteration t and layer k of the NN :

- We find the gradients dw^k, db^k

the exponential moving average is introduced as:

$$V_{dw^k} = \beta \times V_{dw^k} + (1 - \beta) \times dw^k$$

Similarly:

$$V_{db^k} = \beta \times V_{db^k} + (1 - \beta) \times db^k$$

Hence Weight updation with learning rate α becomes:

$$w^k = w^k - \alpha \times V_{dw^k}$$
$$b^k = b^k - \alpha \times V_{db^k}$$

Note: This Optimizer is called Gradient Descent with Momentum

How to further reduce the oscillations of Gradient Descent ?

Ans: optimizer tends to move in direction (oscillations) when gradient of one weight is greater than the other
- Meaning Δb >>> Δw

Hence to reduce this moving direction:

$$V_{dw^k} = \beta V_{dw^k} + (1 - \beta)(dw^k)^2$$
$$V_{db^k} = \beta V_{db^k} + (1 - \beta)(db^k)^2$$

And weight updation becomes:

$$w^k = w^k - \alpha \times \frac{dw^k}{\sqrt{V_{dw^k} + \epsilon}}$$
$$b^k = b^k - \alpha \times \frac{db^k}{\sqrt{V_{db^k} + \epsilon}}$$

where ϵ is a very small value = 10^{-8}

How is squaring useful ?

Ans: as gradients in which the optimizer moves is higher then:

- the square of the gradient will be much high
- thus making $V_{db^k} > V_{dw^k}$ and $\frac{1}{V_{db^k}} < \frac{1}{V_{dw^k}}$
- Therefore after weight updation, w^k reaches optimal value faster

Note: This approach is known as RMSprop

Is there a way to combine both RMSprop's decreased oscillation and Momentum fast convergence ?

Ans: This is done by Adam which defines

- Momentum:

$$V_{dw^k} = \beta_1 V_{dw^k} + (1 - \beta_1)\, dw^k$$
$$V_{db^k} = \beta_1 V_{db^k} + (1 - \beta_1)\, db^k$$

- RMSprop:

$$S_{dw^k} = \beta_2 S_{dw^k} + (1 - \beta_2)\, (dw^k)^2$$
$$S_{db^k} = \beta_2 S_{db^k} + (1 - \beta_2)(\, db^k)^2$$

Now both in RMSprop and Momentum, the initial averaged out values are biased,

- so to kickstart the algorithm Biasness correction is done such that:

$$V_{dw^k}^{Corrected} = \frac{V_{dw^k}}{1-\beta_1^{\,t}}$$
$$V_{db^k}^{Corrected} = \frac{V_{db^k}}{1-\beta_1^{\,t}}$$

$$S_{dw^k}^{Corrected} = \frac{S_{dw^k}}{1-\beta_2^{\,t}}$$
$$S_{db^k}^{Corrected} = \frac{S_{db^k}}{1-\beta_2^{\,t}}$$

Therefore Weight updation becomes:

$$w^k = w^k - \alpha \times \frac{V_{dw^k}^{Corrected}}{\sqrt{S_{dw^k}^{Corrected}} + \epsilon}$$
$$b^k = b^k - \alpha \times \frac{V_{db^k}^{Corrected}}{\sqrt{S_{db^k}^{Corrected}} + \epsilon}$$

Hyperparameter tuning

How to reduce the overfitting of NN and improve performance?

1. Regularization: Regularization cannot be done in NN
   - due to L layered weight matrix $W = [W^1, W^2, W^3, \ldots\ldots, W^L]$

   Hence a hack is used called Forbenius Norm

$$Reg = \frac{\lambda}{2n}\sum_{k=1}^{k=L} ||W^k||_F^{\ 2}$$

Where n is the number of samples and k is the current layer

we define $||W^k||_F^{\ 2}$ as:

$$||W^k||_F^{\ 2} = \sum_{i=1}^{n^{k-1}}\sum_{j=1}^{n^k} (w_{ij}^{\ k})^2$$

Where , n^k is the number of neurons in the current layer k and n^{k-1} is the number of neurons in the previous layer $k-1$

How are the weights updated ?

Ans: Loss is defined as

$$Loss(L) = \frac{1}{2n}\sum_{i=1}^{n} L(y_i, \hat{y}_i) + \frac{\lambda}{2n}\sum_{k=1}^{L} ||W^k||_F^{\ 2}$$

Hence Gradient becomes:

$$\frac{dL}{dW^k} = (From\ Backprop) + \frac{\lambda}{n}W^k$$

Hence weight Updation becomes:

$$w^k = w^k - \alpha(From\ Backprop) - \alpha\frac{\lambda}{n}W^k$$

$$w^k = (1 - \alpha\frac{\lambda}{n})w^k - \alpha(From\ Backprop)$$

Note: the extra $(1 - \alpha\frac{\lambda}{n})$ is known as weight decay

2. Dropout: Regularizes the NN by :
   - Dropping weights(Edges) of the NN
   - By creating a mask (d^k) through a random probability matrix ($P(W^k)$) for the k^{th} layer such that:

$$Mask\ (d^k)\ =\ 1\ if\ P(W^k)\ >\ dropout\ rate\ (r)$$
$$else\ Mask\ (d^k)\ =\ 0$$

   - During test time, all the weights are upscaled by a factor of $p\ =\ 1\ -\ r$

What is the need of Upscaling of weights during test time ?
Ans: Weight updation do not take place for the ones which are dropped
   - Making the weights not reach its optimal values
   - Hence for optimal values, upscaling during test time is done

Note: During Test time, no Dropout takes place

3. Batch normalization: Standardizing the input is one of the important steps for reaching global minima
   - And since computing activation functions, weight multiplications and biases, the input to hidden layers tend to have different distributions
   - These changed distributions gets amplified as we go down the layers of NN
   - This is known as Internal Covariate Shift

Hence data standardization is performed as :

$$mean\ (\mu)\ =\ \frac{1}{m}\sum_{i=1}^{m} z_i$$

$$\text{Variance } (\sigma^2) = \frac{1}{m}\sum_{i=1}^{m} (z_i - \mu)^2$$

$$Z_{norm}\ =\ \frac{(z_i-\mu)}{\sqrt{\sigma^2+\epsilon}}$$

where m is the number of neuron in a layer and $\epsilon\ =\ e^{-10}$

Is having normal distribution for all layers a good thing ?

Ans: No, since two layers having the exact same mean and variance, makes 2nd layer redundant , therefore :

$$\hat{Z}\ =\ \gamma\ \times Z_{norm}\ +\ \beta$$

Where γ, β becomes two learnable parameters

4. Early Stopping: Sometimes the NN performance increases for a certain epoch and decreases on later training epochs
    - So in order to prevent the model to update weights on the later training epochs
    - The weights of the best validation score model is stored using `ModelCheckpointCallback`
    - After which the model runs for a certain threshold after which training stops
    - This stopping of training is done through another callback using `EarlyStoppingCallback`

5. LearningRateDecay: Sometimes model gets stuck around the global minima,
    - due to high learning rate

    And be taking alot of epochs to reach global minima
    - When learning rate is quite small

    Hence to make the NN train faster with high accuracy,
    - An adaptive Learning rate is used such that
    - The learning rate is reduced gradually over epochs
    - So the NN first quickly reaches around the global minima
    - Then converges to global minima with a smaller learning rate
    - This is implemented using a callback `LearningRateScheduler`

Practical Aspects

What are all the hyperparameters for NN ?

1. Number of epochs
2. NN depth and complexity
3. Batch Normalization
4. Learning rate
5. Regularization
6. Dropout
7. Choosing optimizer
8. Collecting more data

Note: Collecting data should be the last resort, since
- Data in general is hard to get
- Collecting data is time intensive
- Does not guarantee an improved performance of NN

What order should the major hyperparameters be tuned for NN ?

1. Learning Rate
2. β value of GD with Momentum
3. Number of Hidden units/ Neurons
4. Batch size
5. Number of layers of NN
6. Learning Rate Decay
7. β_1, β_2, ϵ of Adam

With such a variety of hyperparameters to experiment with,

- it becomes very important to know which hyperparameter to tune to enhance model performance
- This is known as Orthogonalization of NN

What to tweak if NN has a bad training performance ?
Ans: Clearly NN underfits hence:

- More epochs
- Deeper and Complex NN
- Different Optimizer
- More data

What to tweak if NN has good training performance but bad validation performance ?
Ans: Clearly NN underfits hence:

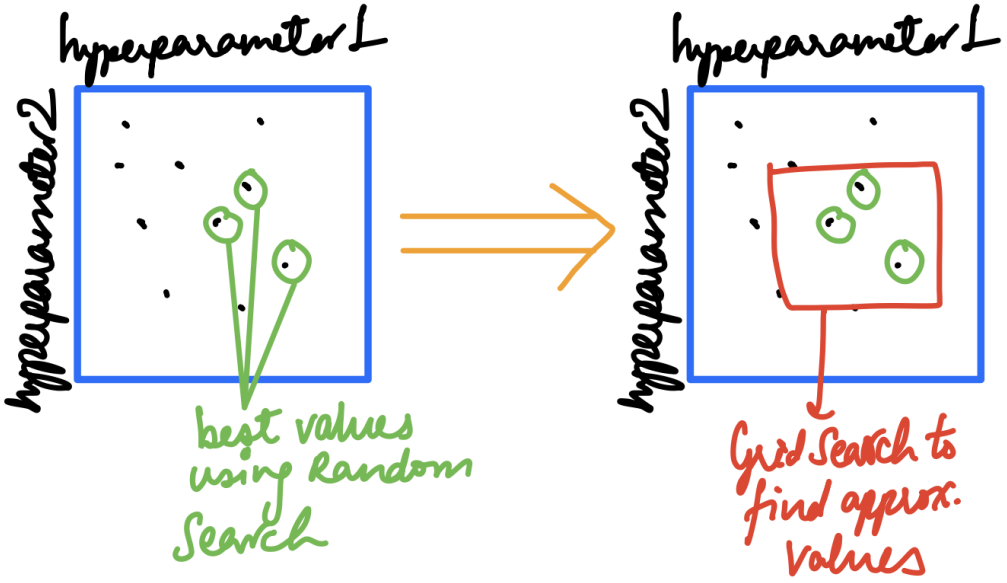
- Use simple NN
- Regularization
- Dropout
- Batch Normalization
- Diverse training samples

What to tweak if NN has bad testing performance but good training and validation performance ?
Ans: Though its not a good practice to tune NN for test data, yet some tweaks are:

- Changing loss function
- More Validation data

How to find the correct value for the hyperparameters ?
Ans: Perform Random Search and get some hyperparameter value ranges

- Followed by Grid Search to have more accurate results



Before Hyperparameter tuning, its very important to have an error analysis done of the model

- Hence its a good practice to have a human performance on the task
- Along with the maximum attainable performance known as Bayes Optimal error

Why need Human performance ?
Ans: Helps identifying whether model is having

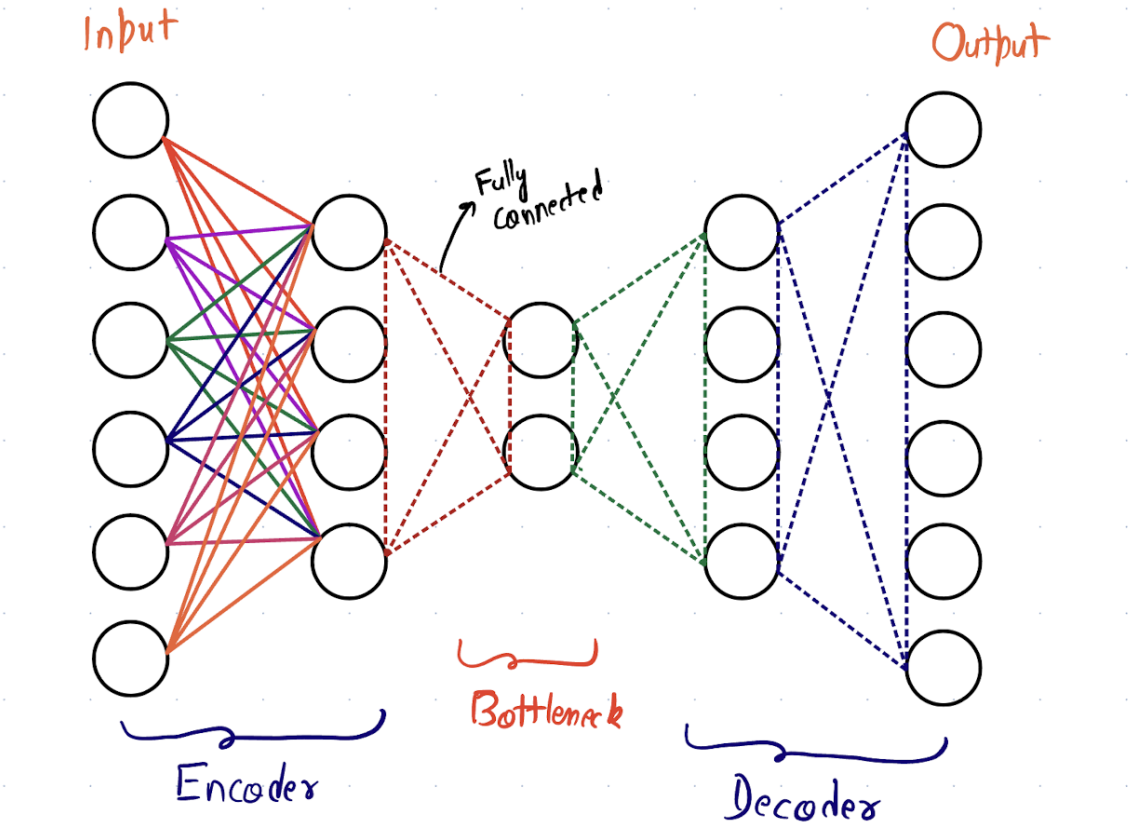
- a low bias and high variance
- or a high variance and low bias

Note: The difference/gap between Human and Training error is called Avoidable Bias

Autoencoders

What are autoencoders (AE)?

- Autoencoders are self supervised learning method where the input (x_i) is same as output ($\hat{x}_i$)
- The network comprises of 3 parts:
    - **Encoder**
    - **Bottleneck layer** => used as encoding/embedding
    - **Decoder**



    - The encoder part compresses the input to lower dimensionality embedding/encoding
    - The decoder produces output by expanding this lower dimensionality embedding and tries to reconstruct input from it.
    - **Goal:** $x_i \sim \hat{x}_i$

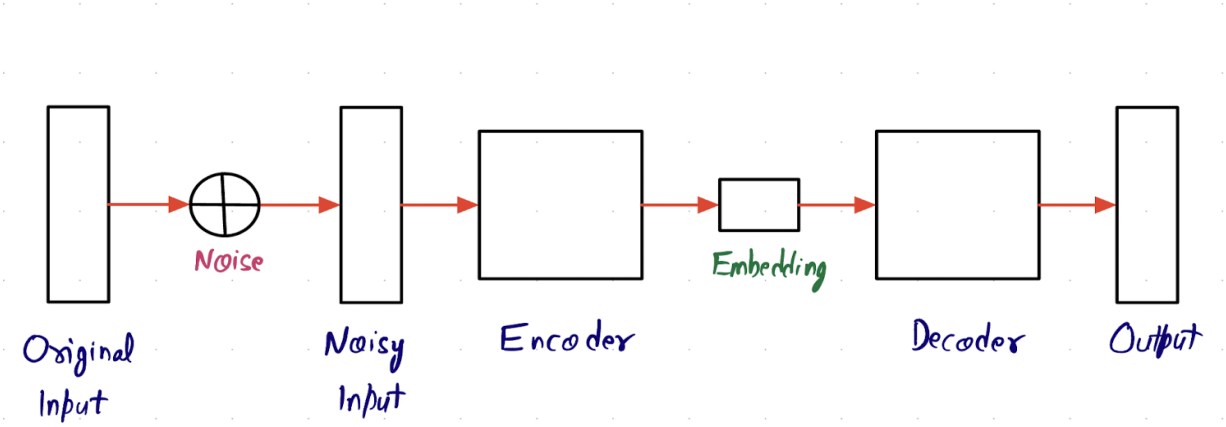
Do note: It is also called unsupervised learning as they don't need labels to train on.

Applications

- **Dimensionality Reduction**

- We can use AE to reduce the dimensionality of the data.
- Do note that
  - The dim. Reduction is data specific
    - For example: If the AE has been trained on handwritten digits, we can't expect it to compress cats and dogs images.
    - It'll be able to meaningfully compress data similar to what it has been trained on
  - The output of the decoder will not be exactly the same as input i.e. there'll be loss of information.
- **Code:** Link

- **Denoising AE**
  - In order to make sure that AE doesn't overfit i.e. it doesn't simply learn to copy input to output
    - We add random noise to that data



What happens when we add random noise?

- If AE recreates the noisy input, it means it has overfitted
- Think of it as regularizing the data
  - As there is no pattern to noise, network shouldn't recreate it
- **Code:** Link

- **Recommender Sys using AE**
  - We can use AE to generate embeddings to find similar items (i.e. as a Recommender System)
  - In order to do so
    - We feed sparse data as input to the network

        - Learn the dense embeddings
        - Find similar items using dense embeddings
    - For example:
        - Find similar movies for a given user-item interaction matrix
            - We feed movies vector (item vector) as input to AE
            - The network learns the dense embeddings.
            - Using cosine similarity on these embeddings, we find similar movies (higher the score, more similar the movie).
    - **Code: Link**

Is it necessary for the encoder and decoder to be symmetric ?

- Not necessarily.
- Earlier we used to keep them symmetric i.e. k^{th} and $n - k^{th}$ layer will have same number of neurons

  Why did we keep the network symmetric ?
    - It was because of weights sharing (weights tying)
    - Weights were shared between encoder and decoder
    - So as to reduce the number of parameters.