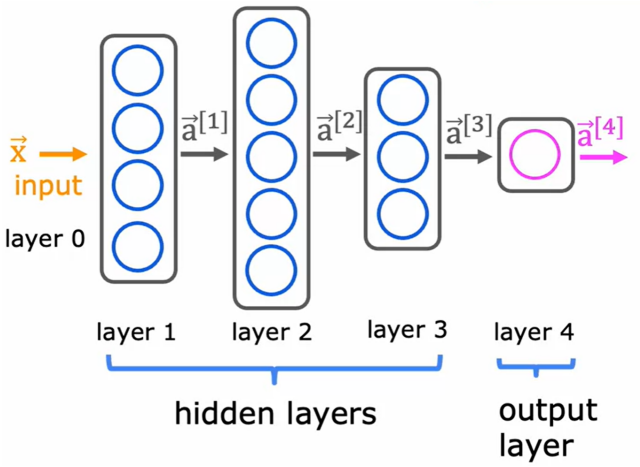
Neural networks intuition

Neural networks are layers of interconnected neurons. The three main types of layers are input, hidden, and output layers. The input layer receives input data, the hidden layers process this data, and output layer produces the final result.



The following equation can help you build a better neural network intuition:

$$a_j^{[l]} = g(\vec{w}_j^{[l]} \cdot \vec{a}^{[l-1]} + b_j^{[l]})$$

Here are some explanations of the equation above:

- $a_j^{[l]}$: the activation value of layer l, unit(neuron) j
- $\vec{w}_j^{[l]}$ and $b_j^{[l]}$: the weights and bias of layer l, unit(neuron) j
- $\vec{a}^{[l-1]}$: the output of layer $l-1$ (previous layer)
- $g(\dots)$: sigmoid function, or **activation function** in neural network

Forward-propagation

Forward propagation is the process through which input data is passed through the neural network to generate an output. The following code shows the implementation of this algorithm:

```
def dense(a_in, W, b):
    """
    Computes dense layer
    Args:
      a_in (ndarray (n, )) : Data, 1 example
      W    (ndarray (n,m)) : Weight matrix, n features per unit, m
units
      b    (ndarray (m, )) : bias vector, m units
```

```
    Returns
      a_out(ndarray (m, )) : m units
    """
    m = W.shape[1]
    a_out = np.zeros(m)
    print(a_in)
    for i in range(m):
        w = W[:, i]
        tmp = np.dot(w, a_in) + b[i]
        a_out[i] = g(tmp) # activation function
    return a_out
```

The process can also be vectorized. It is faster because it takes advantage of specialized hardware capabilities. Assume that the size of input matrix is $1 \times n$ and the size of the output matrix is $1 \times m$:

```
def dense(a_in, W, b):
    # (1,n) * (n,m) + (1,m) = (1,m)
    z = np.matmul(a_in, W) + b
    a_out = g(z)
    return a_out
```

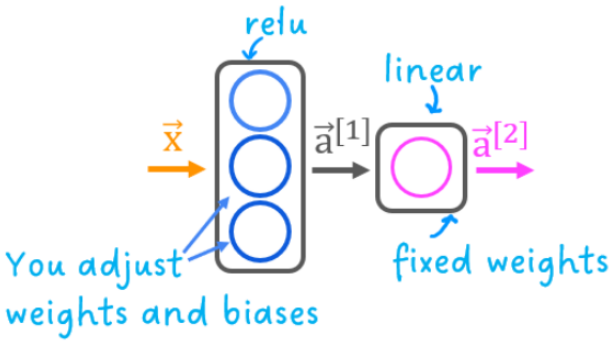
ReLU activation

If you compose two or more linear functions, the result will still be a linear function. It turns out that if the hidden layers are all linear, the result will be equivalent to the linear regression. So we need the activation function to introduce non-linearity to the neutral network.

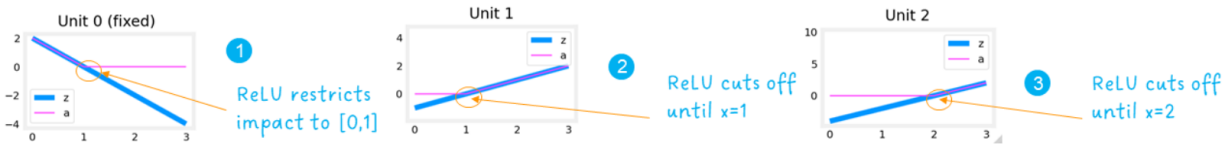
ReLU, or Rectified Linear Unit, is one of the most commonly used activation functions in artificial neural networks. ReLU has a very simple form:

$a = \max(0, z)$

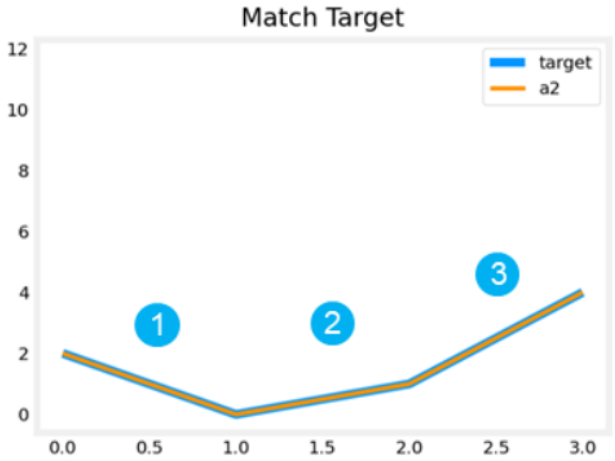
Let's appreciate how ReLU introduce non-linearity to the neutral network. For example:



Assume that $a_0^{[2]} = a_0^{[1]} + a_1^{[1]} + a_2^{[1]} + 0$, and we draw the line of each unit :



Add them up and get the line of the output layer:



Softmax

The softmax function, a generalization of logistic function, can be used for multiclass classification. Given an input vector $z = (z_1, z_2, \dots z_n)$, the softmax function computes the following probabilities for each element z_i :

$$P(z_i) = \sum_{i=1}^{n} \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$$

After applying softmax, each output will be between 0 and 1 and the outputs will add to 1, so that they can be interpreted as probabilities for each category.

The loss function associated with Softmax, the cross-entropy loss, is:

$$L(\mathbf{a}, y) = \begin{cases} -\log a_1 & y = 1 \\ \dots & \\ -\log a_n & y = n \end{cases}$$

Now the cost is:

$$J(\mathbf{w}, b) = -\frac{1}{m} \sum_{i=1}^{m} \sum_{j=1}^{N} [y^{(i)} = j] \cdot \log\left(\frac{e^{z_j^{(i)}}}{\sum_{k=1}^{N} e^{z_k^{(i)}}}\right)$$

When it comes to softmax implementation in TensorFlow, we have a more numerically accurate way to do it:

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(25, activation = 'relu'),
    tf.keras.layers.Dense(15, activation = 'relu'),
    tf.keras.layers.Dense(4, activation = 'linear') # raw logits
])
model.compile(
    loss =
tf.keras.losses.SparseCategoricalCrossentropy(from_logits = True),
    optimizer = tf.keras.optimizers.Adam(0.001)
)
```

The `from_logits` parameter controls whether the input to the softmax function is already in the form of logits. When `from_logits = Ture`, the softmax is applied to raw logits. It gives TensorFlow the ability to rearrange terms and compute softmax in a more numerically accurate way.

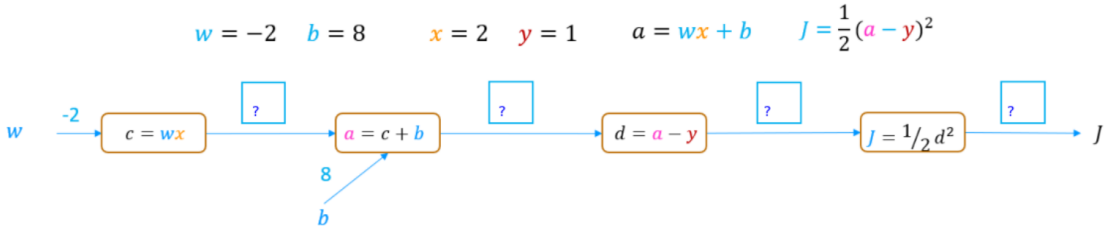
Adam

Adam(Adaptive moment estimation), is an iterative optimization algorithm that computes adaptive learning rate. The algorithm works like this:

- If w keep moving in the same direction, increase the learning rate.
- If w keep oscillating, reduce the learning rate.

Back-propagation

Consider a computation graph of a simple neural network:



According to what we learn in course one, we can apply forward-propagation to the graph and compute all the value we need. Let's put the process in reverse to rectify our parameters by gradient descent.

We can compute the derivative term:

$$\frac{\partial J}{\partial d} = d = 3$$

$$\frac{\partial d}{\partial a} = 1$$

$$\frac{\partial a}{\partial c} = 1$$

$\frac{\partial c}{\partial w} = x = 2$

Back-propagation use chain-rule to optimize the calculation of gradient descent:

$\frac{\partial J}{\partial a} = \frac{\partial J}{\partial d} \cdot \frac{\partial d}{\partial a} = 3$

$\frac{\partial J}{\partial c} = \frac{\partial J}{\partial a} \cdot \frac{\partial a}{\partial c} = 3$

$\frac{\partial J}{\partial w} = \frac{\partial J}{\partial c} \cdot \frac{\partial c}{\partial w} = 6$

After we get the partial derivative term, we can rectify the parameter just by multiplying the learning rate.

Model Evaluation

It is common to split your data into three parts:

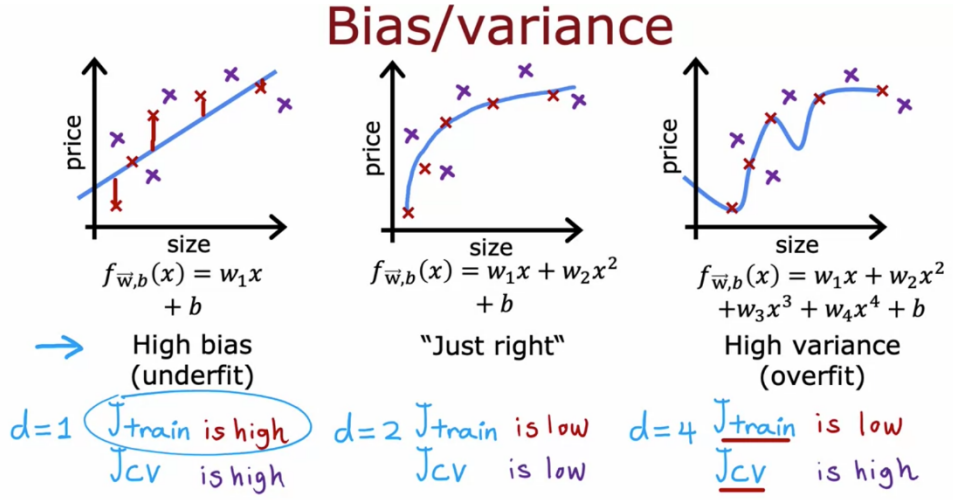
- **training set** - used to train the model.
- **cross validation set** - used to evaluate the different model configurations you are choosing from. You can use this to make a decision on what polynomial features to add to your dataset.
- **test set** - used to give a fair estimate of your chosen model's performance against new examples. This should not be used to make decisions while you are still developing the models.

When studying this part, I am confused about why we used cross validation set. Suggest that we only use test set and select the best d (the dimension of the feature) that has minimum $J_{test}(w^{<d>}, b^{<d>})$. It turns out $J_{test}(w^{<d>}, b^{<d>})$ will be an optimistic estimate of generalization error(ie. $J_{test}(w^{<d>}, b^{<d>}) <$ generalization error).

To give a further explanation, let's imagine that five models have the same generalization error. But since an extra parameter d was chosen using the test set, $J_{test}(w^{<d>}, b^{<d>})$ will be the minimum of five variables(randomly distribute around the generalization error). So it can't give a fair estimate of the generalization error. We should use cross validation set to ensure that test set is not under the impact of selection.

Bias and variance

The training and cross validation errors can tell you what to try next to improve your models. Specifically, it will show if you have a high bias (underfitting) or high variance (overfitting) problem.



Before you can diagnose a model for high bias or high variance, it is usually helpful to first have an idea of what level of error you can reasonably get to. You can check the human level performance or competing algorithm's performance.

To fix a high bias problem, you can:

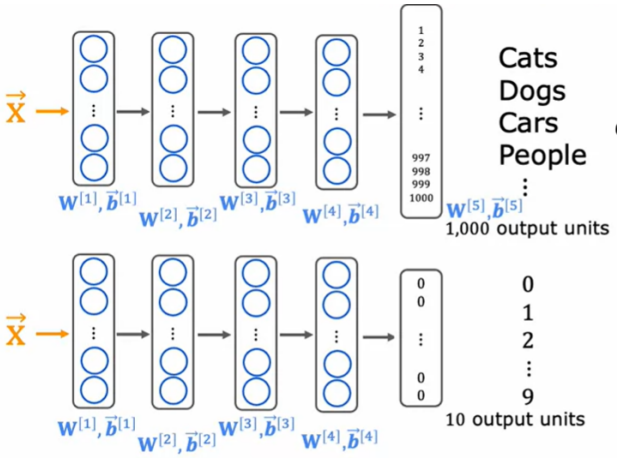
- try adding polynomial features
- try getting additional features
- try decreasing the regularization parameter

To fix a high variance problem, you can:

- try increasing the regularization parameter
- try smaller sets of features
- get more training examples

Transfer learning

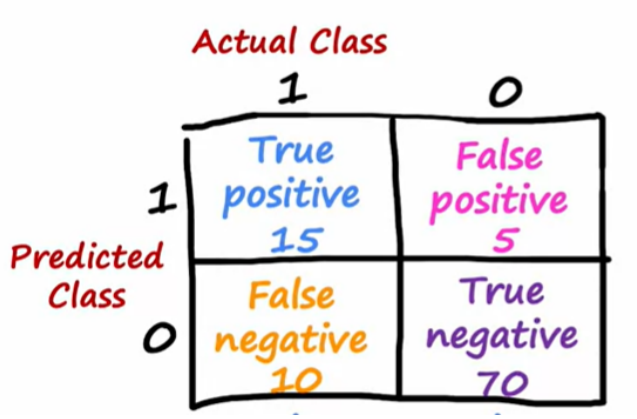
Transfer learning is a machine learning technique where a model trained on one task is adapted for a second related task. Instead of training a model from scratch for a new task, we can take the advantage of pre-trained model.



After trained on a large dataset for a specific task, the early layers of the pre-trained model captures general features. We can replace the output layer and use some data to refine the model. So we can transfer the pre-trained model to achieve our goal.

Skewed dataset

What is skewed dataset? For rare disease classification, we find that you've got 1% error on the test set. Only 0.5% of patients have the disease. So an algorithm which print zero has 99.5% accuracy. The fact indicates that we can't use accuracy to evaluate our model. We need some other metrics.



True positive means that we predict positive for the data that is supposed to have the disease)

False negative means that we predict negative for the data that is supposed to have the disease.

The False positive and true negative both have the similar meaning. We can define the metrics "precision" and "recall" as following:

$$precision = \frac{True\ positive}{True\ pos + False\ pos}$$

$$recall = \frac{True\ positive}{True\ pos + False\ neg}$$

So the algorithm printing zero has both $precision = 0$ and $recall = 0$, which is not a good algorithm under our constraints. There comes a question that if we have different precision and recall, how can we trade off the two metrics. We can further define F1 score to select the best algorithm:

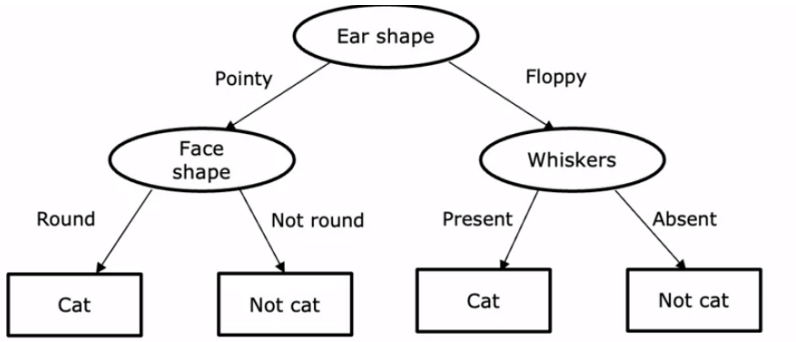
$$F1\ score = 2\frac{precision \cdot recall}{precision + recall}$$

F1 score is a form of harmonic mean, emphasizing the smaller value more.

Decision Trees

Let's take the cat classification for the example. Suggest we want to classify cat for many categories. We have investigated the features of animals as the training data. How can we build a cat classifier?

It turns out we can build trees, whose node represent the features and leave stand for category. When walking down the tree, you are reaching the ideal category.

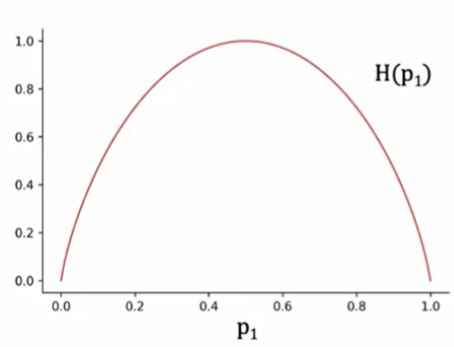


Here comes two questions.

The first question is how to choose what feature to split on at each node during training process. We want to **maximize purity**. In other words, we want to put cat to the left and put others to the right.

Entropy is a measure of impurity. Suggest p_1 equal to the fractions of examples that are cats. Entropy is:

$$H(p_1) = -p_1 \log p_1 - (1 - p_1) \log(1 - p_1)$$



Not that $0 \log 0 = 0$. After splitting, we can compute the entropy reduction. Let me define w^{left} to be the fraction of examples on the left branch. $w^{\text{left}} = \frac{3}{7}$ indicates that we have overall seven animals and three of them are divided to the left branch. The w^{right} is defined in the same way:

$$\text{entropy reduction} = \text{information gain} = H(p_1^{\text{node}}) - (w^{\text{left}} H(p_1^{\text{left}}) + w^{\text{right}} H(p_1^{\text{right}}))$$

The second question is that when do we stop splitting. Keep repeating splitting process until stopping criteria is met:

- When a node is 100% one class

- When splitting a node will result in the tree exceeding a maximum depth
- Information gain from additional splits is less than threshold
- When number of examples in a node is below a threshold

XGBoost

In practice, we would like use multiple trees to increase accuracy. A famous algorithm is called **random forest algorithm**. The algorithm generates an ensemble of trees. During the prediction, we let every tree vote once and the majority of votes will be the final result.

So how can we generate a tree sample? Firstly, we can use sampling with replacement to create new training set of size m. Train a decision tree on the new dataset. Secondly, when choosing a feature to use to split, if n features are available, pick a random subset of $k < n$ features and allow the algorithm to only choose from that subset of features.

XGBoost algorithm improves the performance of random forest algorithm. Instead of picking from all examples with equal $1/m$ probability, make it more likely to pick misclassified examples from previously trained trees.