Siamese Networks

It is a neural network made up of two identical neural networks which are merged at the end. This type of architecture has many applications in NLP.

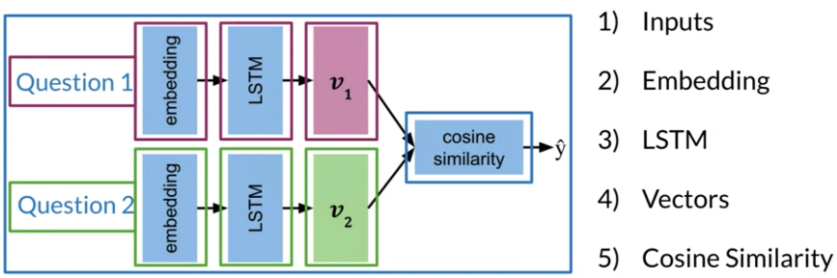
Question Duplicates
How old are you == What is your age
Where are you from != Where are you going

What do Siamese Networks learn?
Identify similarity between things.

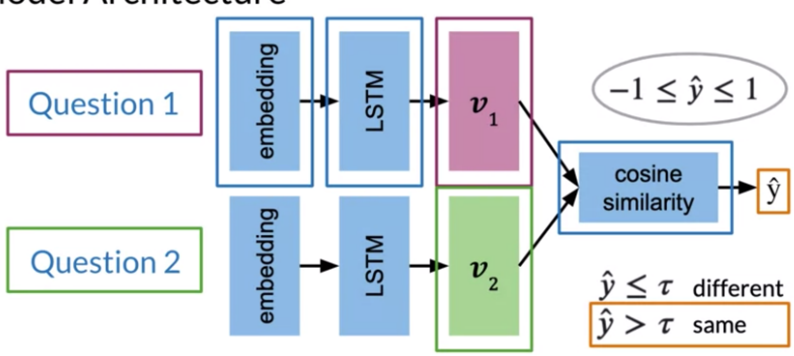
Applications
* Authenticate handwritten. Checks
* Question dupes
* Search engine queries

Architecture
They have two identical sub-networks which are merged together through a dense layer to produce a final output or its similarity score. I like to think of these two sub-networks as sister-networks which come together to produce a similarity score

Model Architecture



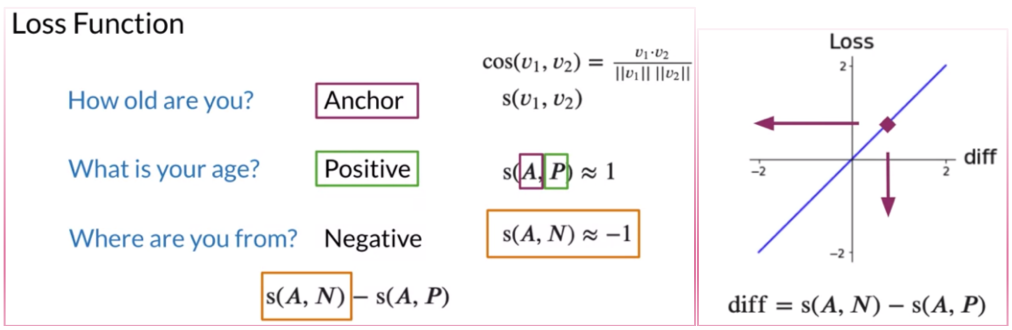
Model Architecture



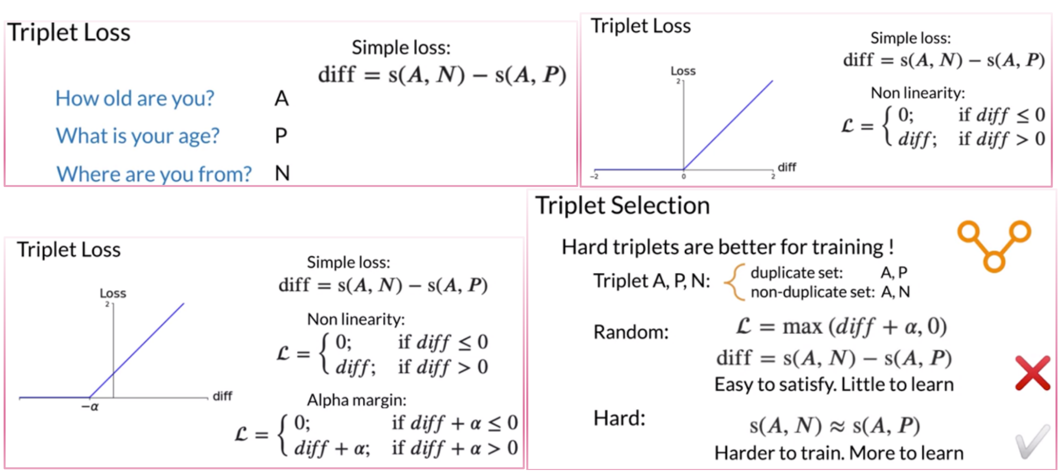

```
Normalize
def normalize(x):
    return x / np.sqrt(np.sum(x * x, axis=-1, keepdims=True))

np.linalg.norm(x, axis=-1, keepdims=True)
vocab_size = 500
model_dimension = 128
```

```
# Define the LSTM model
LSTM = tl.Serial(
        tl.Embedding(vocab_size=vocab_size, d_feature=model_dimension),
        tl.LSTM(model_dimension),
        tl.Mean(axis=1),
        tl.Fn('Normalize', lambda x: normalize(x))
    )

# Use the Parallel combinator to create a Siamese model out of the LSTM
Siamese = tl.Parallel(LSTM, LSTM)

def show_layers(model, layer_prefix):
    print(f"Total layers: {len(model.sublayers)}\n")
    for i in range(len(model.sublayers)):
        print('========')
        print(f'{layer_prefix}_{i}: {model.sublayers[i]}\n')

print('Siamese model:\n')
show_layers(Siamese, 'Parallel.sublayers')

print('Detail of LSTM models:\n')
show_layers(LSTM, 'Serial.sublayers')
```

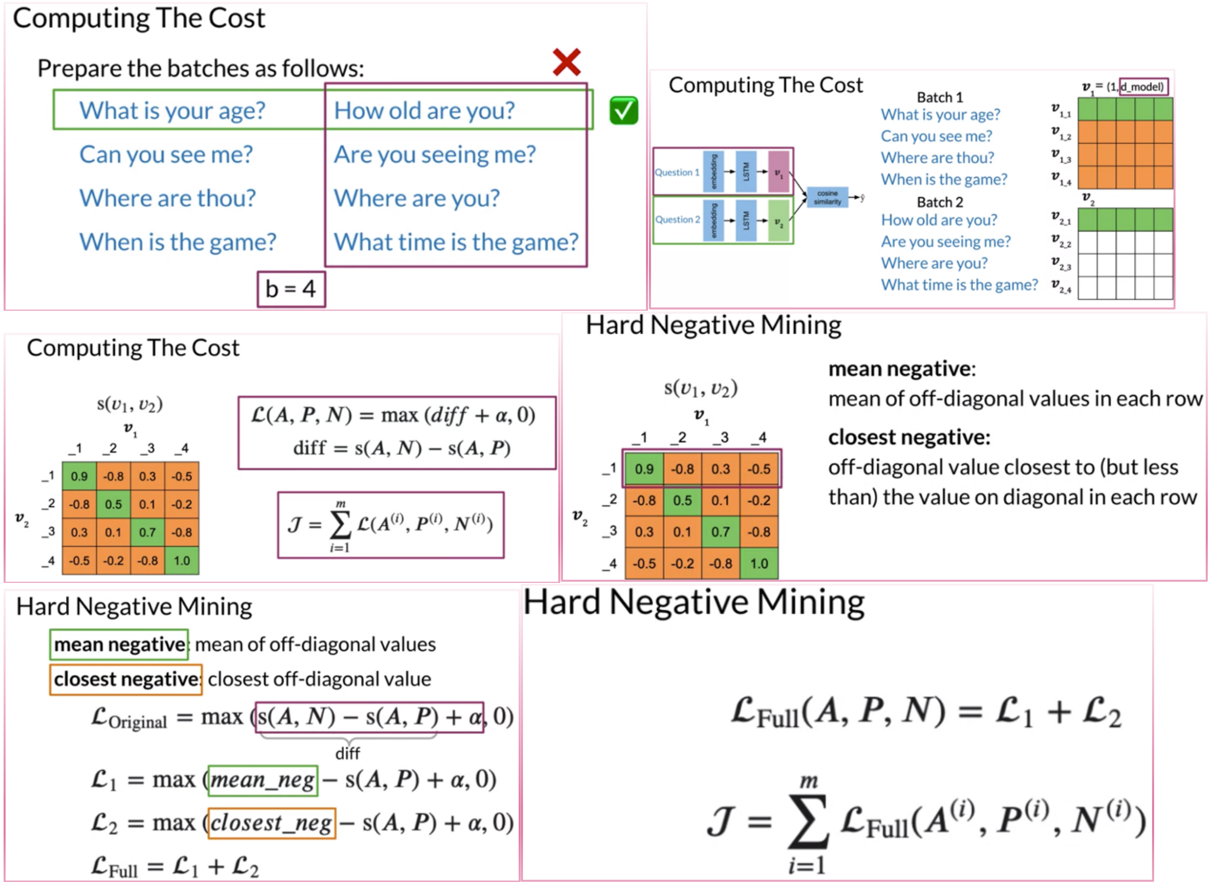
Cost Function



Triplets



Computing the cost



Mean Negative

$mean_neg$mean_neg is the average of the off diagonals, the s(A,N)s(A,N) values, for each row.

Closest Negative

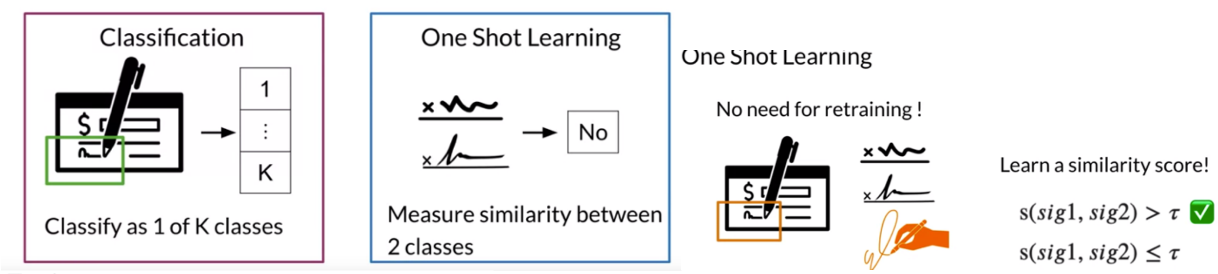
$closest_neg$closest_neg is the largest off diagonal value, s(A,N)s(A,N), that is smaller than the diagonal s(A,P)s(A,P) for each row.

```
# Positives
# All the s(A,P) values : similarities from duplicate question pairs (aka Positives)
# These are along the diagonal
sim_ap = np.diag(sim)
print("sim_ap :")
print(np.diag(sim_ap), "\n")

# Negatives
# all the s(A,N) values : similarities the non duplicate question pairs (aka Negatives)
# These are in the off diagonals
sim_an = sim - np.diag(sim_ap)
print("sim_an :")
print(sim_an, "\n")

print("-- Outputs --")
# Mean negative
# Average of the s(A,N) values for each row
mean_neg = np.sum(sim_an, axis=1, keepdims=True) / (b - 1)
print("mean_neg :")
print(mean_neg, "\n")

# Closest negative
# Max s(A,N) that is <= s(A,P) for each row
```

```
mask_1 = np.identity(b) == 1            # mask to exclude the diagonal
mask_2 = sim_an > sim_ap.reshape(b, 1)  # mask to exclude sim_an > sim_ap
mask = mask_1 | mask_2
sim_an_masked = np.copy(sim_an)         # create a copy to preserve sim_an
sim_an_masked[mask] = -2

closest_neg = np.max(sim_an_masked, axis=1, keepdims=True)

# Alpha margin
alpha = 0.25

# Modified triplet loss
# Loss 1
l_1 = np.maximum(mean_neg - sim_ap.reshape(b, 1) + alpha, 0)
# Loss 2
l_2 = np.maximum(closest_neg - sim_ap.reshape(b, 1) + alpha, 0)
# Loss full
l_full = l_1 + l_2
# Cost
cost = np.sum(l_full)

print("-- Outputs --")
print("loss full :")
print(l_full, "\n")
print("cost :", "{:.3f}".format(cost))
```

One shot learning

Classification vs One Shot Learning



Testing

1. Convert each input into an array of numbers
2. Feed arrays into your model
3. Compare v_1, v_2 using cosine similarity
4. Test against a threshold τ

```
for j in range(batch_size):        # Iterate over each one of the elements in the batch

    d = np.dot(v1[j],v2[j])        # Compute the cosine similarity between the predictions as l2 normalized, ||v1[j]||==||v2[j]||==1 so only dot product is needed
    res = d > threshold            # Determine if this value is greater than the threshold (if it is consider the two questions as the same)
    accuracy += (y_test[j] == res) # Compare against the actual target and if the prediction matches, add 1 to the accuracy

accuracy = accuracy / batch_size   # Divide the accuracy by the number of processed elements
```