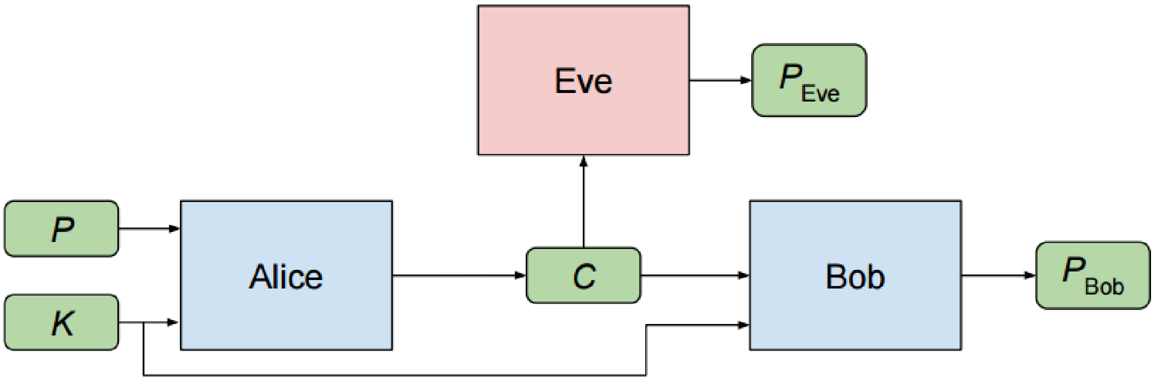
**Project by:** Anand Vibhuti(15MI401), Vertika Sharma(15MI402)

**Objective:** Learning to protect communications by adversarial neural cryptography

**Scenario:** A scenario in security involving three parties: Alice, Bob, and Eve. Typically, Alice and Bob wish to communicate securely, and Eve wishes to eavesdrop on their communications. Eve's goal is simple: to reconstruct P accurately.




**Methodology:**

**1)**In this scenario:

- Alice is the encryption algorithm
- Bob is the decryption algorithm
- Eve is the attacker

  We use 16-bit messages, secret keys and ciphertexts.

**2)**The Alice network will need to have two input vectors: the message to be encrypted, and the secret key. These get concatenated and fed to a dense layer. The signal is then passed through a sequence of four Conv1D layers to create the output.

**3)** The Eve network will have Two dense layers acting on the inputs. The idea is to give it a better chance at decrypting the message, since it doesn't have access to the secret key - it only sees the ciphertext.

**4)** Bob's network is identical to Alice's, except input0 now represents the ciphertext instead of the plaintext.

**5)** The loss for Eve is just the L1 distance between ainput0 and eoutput. Instead of doing an average, sum is taken over all the bits in the message, and its value represents the average number of bits Eve guesses incorrectly. We then take the average across the entire mini-batch with K.mean(). The minimum value of the loss is 0 (Eve guesses all the bits correctly), while the maximum is 16 (Eve is wrong about all the bits).

**6)** First, we want Bob to successfully decrypt the ciphertext.

**7)** We want Alice to learn an encryption scheme which Eve can't break. In an ideal situation, Eve should do random guessing, in which case she would correctly guess half the bits, or m_bits/2 correctly (corresponding to a loss value of 8).

**8)** As per optimizer, we use RMSprop with a default learning rate of 0.001.

**9)** Finally, we create two training models with the loss functions defined earlier.

**10)** Training of the networks.

**11)** Evaluating with providing values.

**Network layers:**

**1)**Alice's Network:

```
[ ] ainput0 = Input(shape=(m_bits,)) # the message
    ainput1 = Input(shape=(k_bits,)) # the key
    ainput = concatenate([ainput0, ainput1], axis=1)

    adense1 = Dense(units=(m_bits + k_bits))(ainput)
    adense1a = Activation('tanh')(adense1)
    areshape = Reshape((m_bits + k_bits, 1,))(adense1a)
    #output of the Dense layer will be 2-dimensional. This needs to be reshaped to (batch_size, m_bits + k_bits, 1) before

    aconv1 = Conv1D(filters=2, kernel_size=4, strides=1, padding=pad)(areshape)
    aconv1a = Activation('tanh')(aconv1)
    aconv2 = Conv1D(filters=4, kernel_size=2, strides=2, padding=pad)(aconv1a)
    aconv2a = Activation('tanh')(aconv2)
    aconv3 = Conv1D(filters=4, kernel_size=1, strides=1, padding=pad)(aconv2a)
    aconv3a = Activation('tanh')(aconv3)
    aconv4 = Conv1D(filters=1, kernel_size=1, strides=1, padding=pad)(aconv3a)
    aconv4a = Activation('sigmoid')(aconv4)

    aoutput = Flatten()(aconv4a)

    alice = Model([ainput0, ainput1], aoutput, name='alice')
```

**2)** Bob's Network:

```
[ ]  binput0 = Input(shape=(c_bits,)) #message
     binput1 = Input(shape=(k_bits,)) #key
     binput = concatenate([binput0, binput1], axis=1)

     bdense1 = Dense(units=(c_bits + k_bits))(binput)
     bdense1b = Activation('tanh')(bdense1)
     breshape = Reshape((c_bits + k_bits, 1,))(bdense1b)

     bconv1 = Conv1D(filters=2, kernel_size=4, strides=1, padding=pad)(breshape)
     bconv1b = Activation('tanh')(bconv1)
     bconv2 = Conv1D(filters=4, kernel_size=2, strides=2, padding=pad)(bconv1b)
     bconv2b = Activation('tanh')(bconv2)
     bconv3 = Conv1D(filters=4, kernel_size=1, strides=1, padding=pad)(bconv2b)
     bconv3b = Activation('tanh')(bconv3)
     bconv4 = Conv1D(filters=1, kernel_size=1, strides=1, padding=pad)(bconv3b)
     bconv4b = Activation('sigmoid')(bconv4)

     boutput = Flatten()(bconv4b)

     bob = Model([binput0, binput1], boutput, name='bob')
```

**3)** Eve's Network:

```
[ ]  einput = Input(shape=(c_bits,)) # the ciphertext

     edense1 = Dense(units=(c_bits + k_bits))(einput)
     edense1a = Activation('tanh')(edense1)
     edense2 = Dense(units=(c_bits + k_bits))(edense1a)
     edense2a = Activation('tanh')(edense2)
     ereshape = Reshape((c_bits + k_bits, 1,))(edense2a)

     econv1 = Conv1D(filters=2, kernel_size=4, strides=1, padding=pad)(ereshape)
     econv1a = Activation('tanh')(econv1)
     econv2 = Conv1D(filters=4, kernel_size=2, strides=2, padding=pad)(econv1a)
     econv2a = Activation('tanh')(econv2)
     econv3 = Conv1D(filters=4, kernel_size=1, strides=1, padding=pad)(econv2a)
     econv3a = Activation('tanh')(econv3)
     econv4 = Conv1D(filters=1, kernel_size=1, strides=1, padding=pad)(econv3a)
     econv4a = Activation('sigmoid')(econv4)

     eoutput = Flatten()(econv4a)# Eve's attempt at guessing the plaintext

     eve = Model(einput, eoutput, name='eve')
```

**Results:**

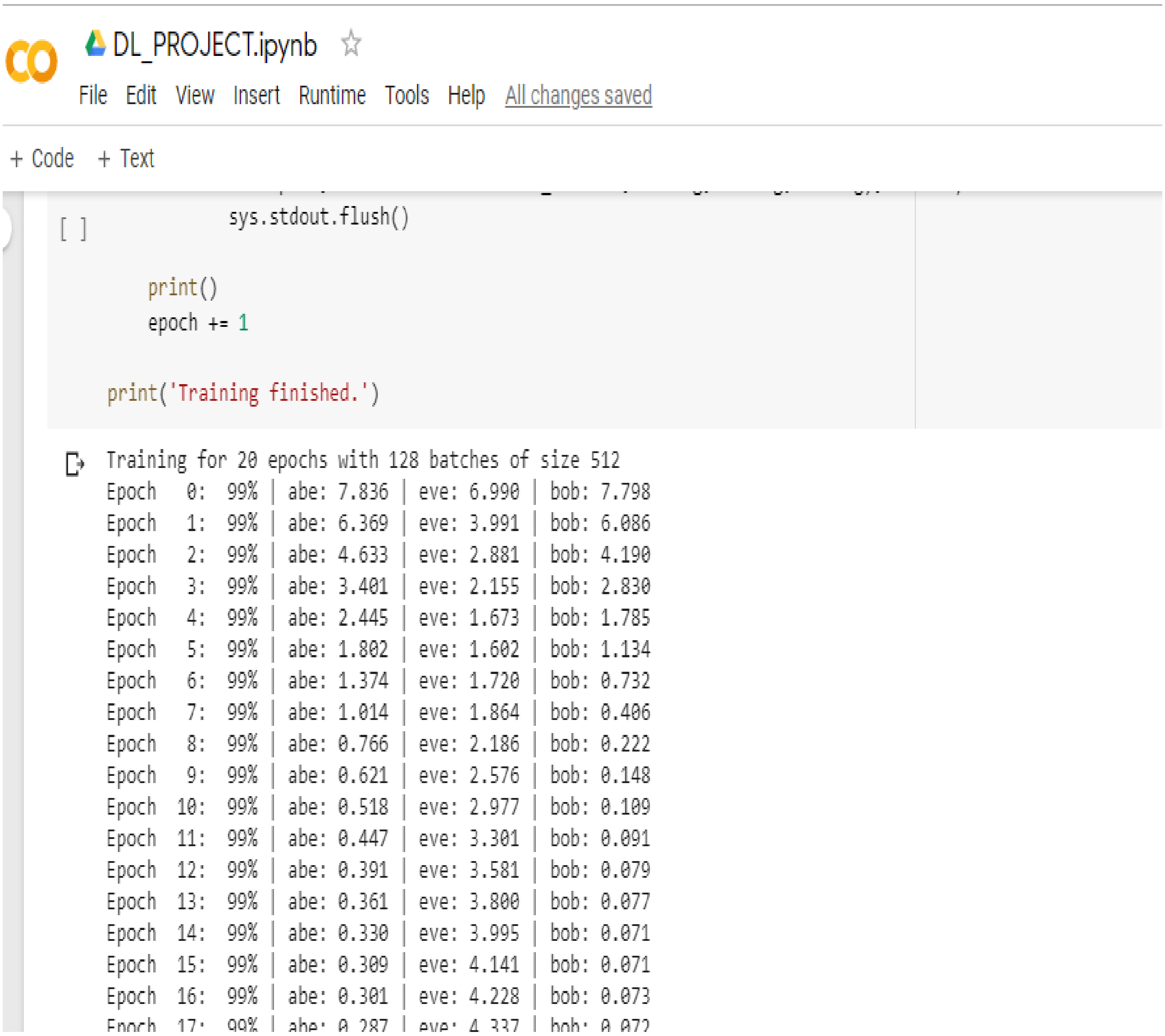



Figure 1: Training the networks

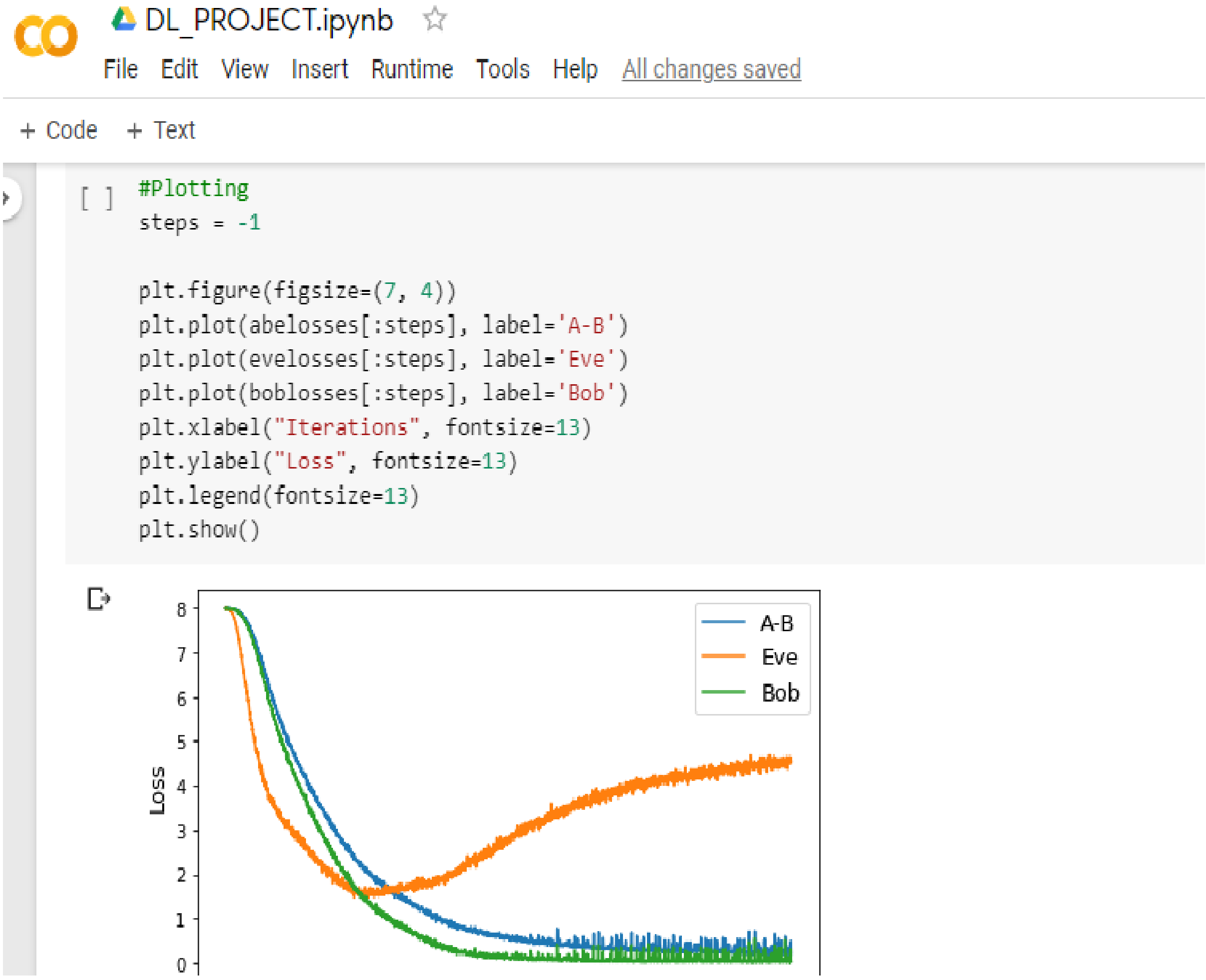



Figure 2: Plots for the computed losses

Reference Paper: **Learning to Protect Communications with Adversarial Neural Cryptography by Martín Abadi, David G. Andersen (Google Brain).**

*(Submitted on 21 Oct 2016)*