Table of Contents

MODULE 1: Introduction to ANN

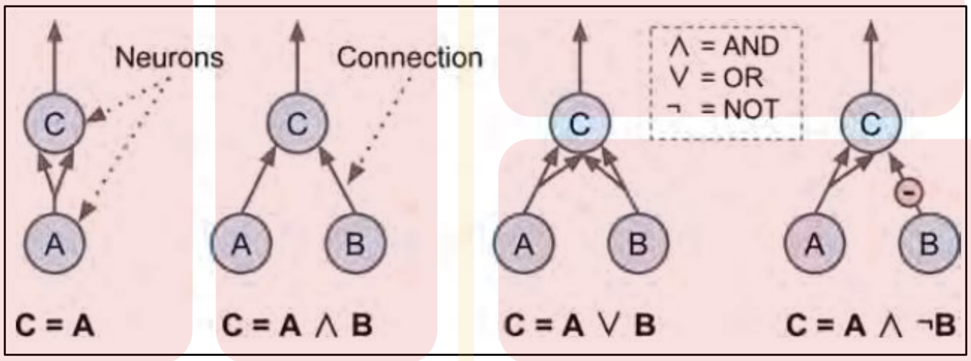
1. Define Neural Networks. Explain how Artificial neuron is related to biological neurons.

A neural network is a machine learning model that mimics the way biological neurons work together to make decisions, enabling tasks such as pattern recognition and decision-making in computer science and artificial intelligence.

- **Biological neurons** receive **short electrical impulses** called signals from other neurons via the synapses.
- When a neuron receives a **sufficient number of signals** from other neurons **within** a few milliseconds, it **fires its own signals**.
- Thus, individual biological neurons seem to behave in a rather simple way, but **they are organized in a vast network of billions of neurons**, each neuron typically connected to thousands of other neurons.
- **Highly complex computations** can be performed by a vast network of fairly simple neurons.
- An **artificial neuron** has **one or more** binary (on/off) **inputs** and **one binary output.**
- The artificial neuron simply **activates its output** when **more than a certain number of its inputs are active.**
- **McCulloch and Pitts** showed that even with such a simplified model it is possible to build a network of artificial neurons that computes any logical proposition you want.

2. With an example explain logical computation with neurons and illustrate how ANNs perform simple logical computation. [July 2023]

Below are ANNs that perform various logical computations **assuming that a neuron is activated when at least two of its inputs are active.**



- The first network on the left is **identity function**: if neuron A is activated, then neuron C gets activated as well **(since it receives two input signals from neuron A),** but if neuron A is off, then neuron C is off as well.
- The second network performs **a logical AND**: neuron C is activated **only when both neurons A and B are activated** (a single input signal is not enough to activate neuron C).
- The third network performs **a logical OR**: neuron C gets activated **if either neuron A or neuron B is activated (or both).**
- Finally, the fourth network computes a slightly more complex logical proposition: **neuron C is activated only if neuron A is active and if neuron B is off**. **If neuron A is active all the time, then you get a logical NOT**: **neuron C is active when neuron B is off, and vice versa.**

3. Describe the perceptron model used in creating artificial neurons.

OR

6. With respect to the perceptron explain the following with a neat diagram. i) Linear Threshold Unit ii) Perceptron Learning Rule [July 2023]

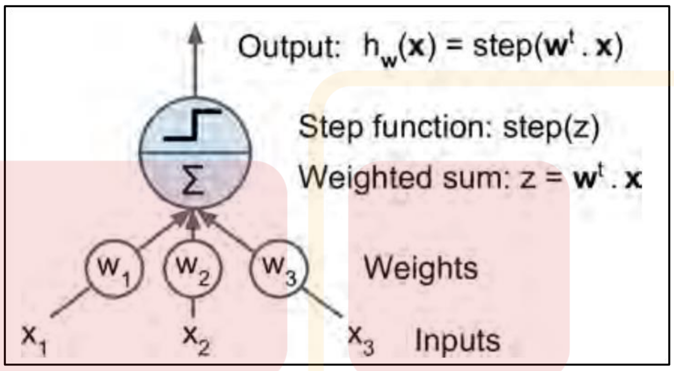
- The Perceptron is based on a slightly different artificial neuron called **a linear threshold unit (LTU):** the **inputs and output are numbers** (instead of binary on/off values) and **each input connection** is associated with a **weight**.
- The **LTU** computes a **weighted sum** of its inputs

$$z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n = w^T \cdot x$$

, then applies a **step function** to that sum and outputs the result:

$$h_w(x) = \text{step}(z) = \text{step}(w^T \cdot x).$$

- **An LTU** can be represented as:

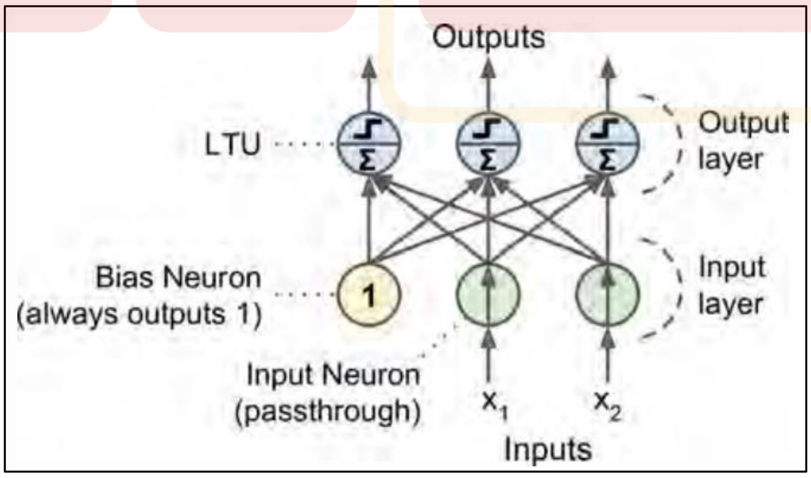


Output: $h_w(\mathbf{x}) = \text{step}(\mathbf{w}^t . \mathbf{x})$
Step function: step(z)
Weighted sum: $z = \mathbf{w}^t . \mathbf{x}$
Weights
w_1
w_2
w_3
x_1
x_2
x_3
Inputs

- The most common step function used in Perceptrons is the **Heaviside step function**. Sometimes the **sign function** is used instead. These functions can be given as follows:

$$\text{heaviside}(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases} \qquad \text{sgn}(z) = \begin{cases} -1 & \text{if } z < 0 \\ 0 & \text{if } z = 0 \\ +1 & \text{if } z > 0 \end{cases}$$

- **A** Perceptron is simply composed **of a single layer of LTUs**, with **each neuron** connected to **all** the **input neurons**.
- **A Perceptron** with two inputs and three outputs is represented below:



- Perceptron learning rule (weight update) is given as:

$$w_{i,j}^{(\text{next step})} = w_{i,j} + \eta(\hat{y}_j - y_j)x_i$$

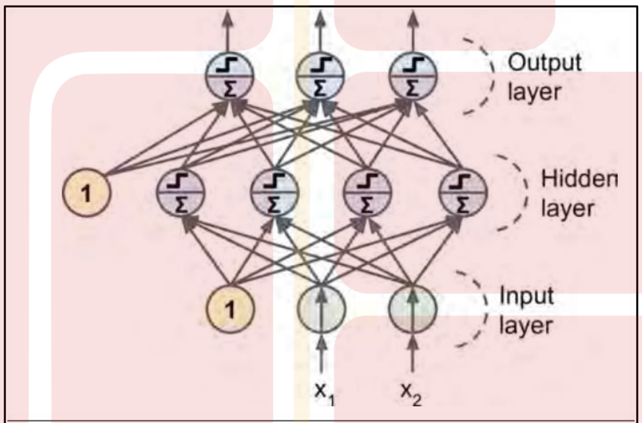
- $w_{i,j}$ is the connection weight between the ith input neuron and the jth output neuron.
- x_i is the ith input value of the current training instance.
- $\hat{y}_j$ is the output of the jth output neuron for the current training instance.
- y_j is the target output of the jth output neuron for the current training instance.
- η is the learning rate.

- Scikit-Learn provides a Perceptron class that implements a single LTU network”

```
from sklearn.linear_model import Perceptron
clf = Perceptron(random_state=42)
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
```

4. Describe the Multilayer Perceptron model along with backpropogation with a neat diagram.

- An MLP is composed of one (passthrough) input layer, one or more layers of LTUs, called hidden layers, and one final layer of LTUs called the output layer. This is shown below:



Multi-Layer Perceptron

- Every layer except the output layer includes a bias neuron and is fully connected to the next layer. When an ANN has two or more hidden layers, it is called a deep neural network (DNN).
- For many years researchers struggled to find a way to train MLPs, without success. But in 1986, a groundbreaking article was published that introduced the **backpropagation training algorithm**.
- Today we would describe it as **Gradient Descent using reverse-mode autodiff**.
- **For each training instance**, the algorithm feeds it to the network and **computes the output of every neuron in each consecutive layer** (this is the **forward pass**, just like when making predictions).
- Then it **measures the network’s output error** (i.e., the difference between the desired output and the actual output of the network), and it **computes how much each neuron** in the **last hidden layer contributed** to **each output neuron’s error**.
- It then **proceeds** to measure how much of these error contributions came from each neuron in the **previous hidden layer and so on** until the algorithm **reaches** the **input layer.**

- **This reverse pass efficiently measures the error gradient across all the connection weights** in the network by **propagating the error gradient backward** in the network hence the name of the algorithm.
- The **last step** of the backpropagation algorithm is a **Gradient Descent step on all the connection** weights in the network, using the error gradients measured earlier. In other words, **it slightly tweaks the connection weights to reduce the error.**

5. With code snippet explain gradient descent method using Tensorflow. [July 2023]

- **USING AUTODIFF:**
  - TensorFlow's **autodiff feature** can automatically and efficiently compute the gradients:

```
gradients = tf.gradients(mse, [theta])[0]
```

  - The **gradients() function** takes the **MSE** and a list of variables (in this case just **theta**), and it creates a **list of MSE** (one per variable) to compute the **gradients of the MSE** with regards **to each variable**.
  - So the **gradients node** will compute the **gradient vector** of the MSE with regards to theta.
- **USING OPTIMIZER:**
  - TensorFlow also provides **a number of optimizers** out of the box, including **a Gradient Descent optimizer.**

```
optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(mse)
```

7. With respect to training a DNN using plain Tensor Flow explain construction phase and execution phase. [July 2023]

- The first step is the **construction phase, building the TensorFlow graph**. The second step is the **execution phase**, where you actually **run the graph to train the model.**
- **CONSTRUCTION PHASE:**
  - we create **placeholders** for the inputs and the targets:

```
X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
y = tf.placeholder(tf.int64, shape=(None), name="y")
```

  - we create a function to build a neuron layer:

```
def neuron_layer(X, n_neurons, name, activation=None):
  with tf.name_scope(name):
    n_inputs = int(X.get_shape()[1])
    stddev = 2 / np.sqrt(n_inputs)
    init = tf.truncated_normal((n_inputs, n_neurons), stddev=stddev)
    W = tf.Variable(init, name="weights")
    b = tf.Variable(tf.zeros([n_neurons]), name="biases")
    z = tf.matmul(X, W) + b
    if activation=="relu":
      return tf.nn.relu(z)
    else:
      return z
```

- we use it to create the DNN:

```
with tf.name_scope("dnn"):
  hidden1 = fully_connected(X, n_hidden1, scope="hidden1")
  hidden2 = fully_connected(hidden1, n_hidden2, scope="hidden2")
  logits = fully_connected(hidden2, n_outputs, scope="outputs", activation_fn=None)
```

- we define the cost function:

```
with tf.name_scope("loss"):
  xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y, logits=logits)
  loss = tf.reduce_mean(xentropy, name="loss")
```

- we create an optimizer:

```
learning_rate = 0.01
with tf.name_scope("train"):
  optimizer = tf.train.GradientDescentOptimizer(learning_rate)
  training_op = optimizer.minimize(loss)
```

- finally we define the performance measure.

```
with tf.name_scope("eval"):
  correct = tf.nn.in_top_k(logits, y, 1)
  accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))
init = tf.global_variables_initializer()
saver = tf.train.Saver()
```

- **<u>EXECUTION PHASE</u>:**
  - Obtain the data:

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/")
```

  - Define the number of epochs that we want to run and the size of the mini-batches:

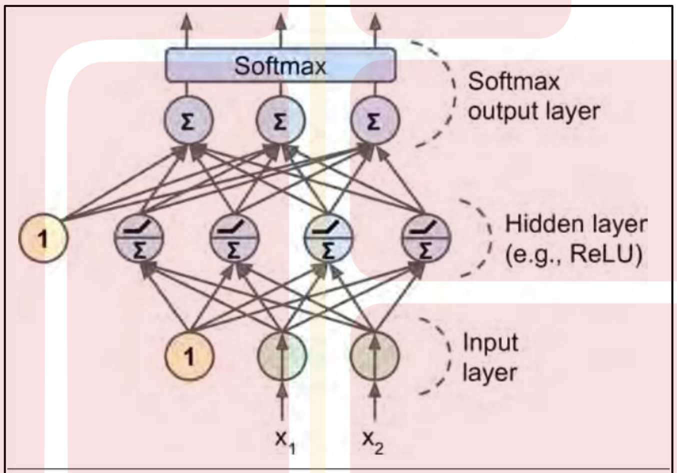
```
n_epochs = 400
batch_size = 50
```

  - Train the model:

```
with tf.Session() as sess:
  init.run()
  for epoch in range(n_epochs):
    for iteration in range(mnist.train.num_examples // batch_size):
      X_batch, y_batch = mnist.train.next_batch(batch_size)
      sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
    acc_train = accuracy.eval(feed_dict={X: X_batch, y: y_batch})
    acc_test = accuracy.eval(feed_dict={X: mnist.test.images,
                                        y: mnist.test.labels})
    print(epoch, "Train accuracy:", acc_train, "Test accuracy:", acc_test)
  save_path = saver.save(sess, "./my_model_final.ckpt")
```

8. What is backpropagation model and how does it work?

- For many years researchers struggled to find a way to train MLPs, without success. But in 1986, a groundbreaking article was published that introduced the **backpropagation training algorithm**.
- Today we would describe it as **Gradient Descent using reverse-mode autodiff**.
- **For each training instance**, the algorithm feeds it to the network and **computes the output of every neuron in each consecutive layer** (this is the **forward pass**, just like when making predictions).
- Then it **measures the network's output error** (i.e., the difference between the desired output and the actual output of the network), and it **computes how much each neuron** in the **last hidden layer contributed** to **each output neuron's error**.
- It then **proceeds** to measure how much of these error contributions came from each neuron in the **previous hidden layer and so on** until the algorithm **reaches** the **input layer.**
- **This reverse pass efficiently measures the error gradient across all the connection weights** in the network by **propagating the error gradient backward** in the network hence the name of the algorithm.
- The **last step** of the backpropagation algorithm is a **Gradient Descent step on all the connection** weights in the network, using the error gradients measured earlier.
- In order for this **algorithm to work properly,** the authors made a **key change to the MLP's architecture:** they **replaced** the **step** function with the **logistic** function, ***$\sigma(z) = 1 / (1 + \exp(-z))$.***
- The backpropagation algorithm may be used with **other activation functions, instead of the logistic function. Two other popular activation functions are**:
    - The **hyperbolic tangent** function: ***$\tanh(z) = 2\sigma(2z) - 1$***
    - The **ReLU** function: ***$\text{ReLU}(z) = \max(0, z)$.***



A modern MLP (including ReLU and softmax) for classification

9. List all the hyperparameters that can tweak in a Neural Network? Explain in detail each hyperparameters.

- The hyperparameters that can be tweaked in an NN are:
    - **Number of Hidden Layers**
    - **Number of Neurons per Hidden Layer**
    - **Activation Function**
- **<u>Number of Hidden Layers</u>:**
    - For many problems, we can just begin with a **single hidden layer** and we will get reasonable results.
    - **deep networks** (more than one hidden layer) have a much **higher parameter efficiency** than shallow ones: they can model **complex functions** using **exponentially fewer neurons** than shallow nets, making them much faster to train.

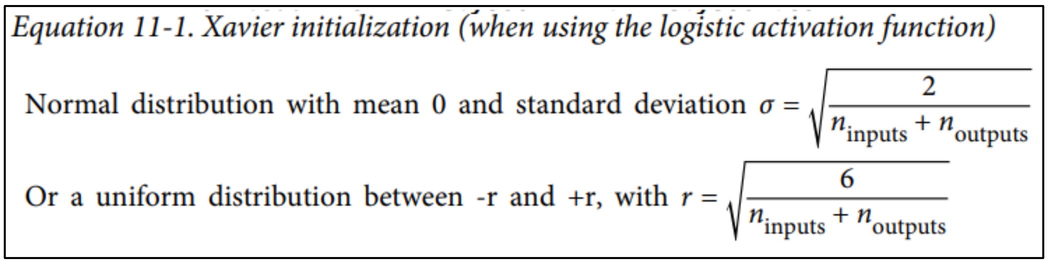
- **Real-world data** is often structured in a **hierarchical way** and DNNs automatically take advantage of this fact: **lower hidden layers model low-level structures**, **intermediate hidden layers combine these low-level structures to model intermediate-level structures**, and the **highest hidden layers and the output layer combine these intermediate structures to model high-level structures.**
- In summary, for many problems we can start with just one or two hidden layers, for more complex problems, we can gradually ramp up the number of hidden layers, until we start overfitting the training set.

- **Number of Neurons per Hidden Layer:**
  - The number of neurons in the input and output layers is **determined by the type of input and output our task requires.**
  - For example, the **MNIST task** requires **28 x 28 = 784 input neurons** and **10 output neurons.**
  - As for the hidden layers, a common practice is to **size them to form a funnel, with fewer and fewer neurons at each layer**— the rationale being that **many low-level features** can **coalesce** into **far fewer high-level features**.
  - A simpler approach is to pick a model with **more layers and neurons than we actually need**, then use **early stopping to prevent it from overfitting**. This has been dubbed the "**stretch pants**" approach.
- **Activation Function:**
  - In most cases we can use the **ReLU activation function in the hidden layers**. It is a **bit faster** to compute than other activation functions, and **Gradient Descent does not get stuck** as much on plateaus.
  - For the **output layer, the softmax activation function** is generally a good choice for **classification tasks**.
  - **For regression tasks**, we can simply use **no activation function at all**.

MODULE 2: Deep Neural network

1. Define Vanishing/Exploding Gradients Problems. Explain briefly the solutions to overcome from this problem.

- Gradients often get **smaller and smaller** as the algorithm progresses **down to the lower layers**.
- As a result, the Gradient Descent update makes the **lower layer connection weights** virtually **unchanged**, and **training never converges to a good solution**. This is called the "**vanishing gradients problem."**
- In some cases, **the opposite can** happen: the **gradients can grow bigger and bigger**, so many layers **get insanely large weight** updates and the **algorithm diverges**. This is the "**exploding gradients problem"**, which is mostly encountered in recurrent neural networks.
- There are 4 solutions to solving vanishing/exploding gradients problem:
- **Xavier and He Initialization:**
  - The connection weights must be initialized randomly as described in below equations:


Equation 11-1. Xavier initialization (when using the logistic activation function)

Normal distribution with mean 0 and standard deviation $\sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$

Or a uniform distribution between -r and +r, with $r = \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$

    - where n_{inputs} and $n_{outputs}$ are the number of input and output connections for the layer whose weights are being initialized (also called fan-in and fan-out).
- <u>**Nonsaturating Activation Functions:**</u>
    - We use a variant of the ReLU function, such as the leaky ReLU. This function is defined as **$\text{LeakyReLU}_\alpha(z) = \max(\alpha z, z)$**
    - The hyperparameter **α** defines how much the function "**leaks**": it is the slope of the function for **$z < 0$**, and is typically set to **0.01**.
- <u>**Batch Normalization:**</u>
    - The technique consists of **adding** an **operation** in the model **just before the activation** function of **each layer**.
    - Simply **zero-centering** and **normalizing the inputs**, then **scaling and shifting** the **result** using **two new parameters per layer** (one for scaling, the other for shifting).
- <u>**Gradient Clipping:**</u>
    - A popular technique to **lessen the exploding gradients** problem is to **simply clip the gradients during backpropagation** so that **they never exceed some threshold**. This is called Gradient Clipping.

2. Compare and contrast the features of RMS prop algorithm with ADAM algorithm. [July 2023]

- <u>**RMSProp:**</u>
    - **AdaGrad** algorithm **slows** down a **bit too fast** and ends up **never converging** to the **global optimum**.
    - The **RMSProp** algorithm **fixes** this by **accumulating only** the **gradients** from the **most recent iterations** (as opposed to all the gradients since the beginning of training).
    - It does so by **using exponential decay** in the **first step**:

Equation 11-7. RMSProp algorithm

$$1.\quad \mathbf{s} \leftarrow \beta \mathbf{s} + (1-\beta)\nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$$

$$2.\quad \theta \leftarrow \theta - \eta\, \nabla_\theta J(\theta) \oslash \sqrt{\mathbf{s} + \epsilon}$$

    - The **decay rate β** is typically set to **0.9**.
    - **β** is a new hyperparameter, but this default value often works well, so you may not need to tune it at all.
    - TensorFlow has an RMSPropOptimizer class:

```
optimizer = tf.train.RMSPropOptimizer(learning_rate=learning_rate,
                                      momentum=0.9, decay=0.9, epsilon=1e-10)
```

    - This optimizer almost **always performs much better than AdaGrad**. It was the preferred optimization algorithm of many researchers **until Adam optimization came around**.
- <u>**Adam Optimization:**</u>
    - Adam, which stands for **adaptive moment estimation**, **combines** the **ideas** of **Momentum optimization** and **RMSProp**.
    - Just like **Momentum optimization** it keeps **track** of an **exponentially decaying average** of **past gradients**, and just like **RMSProp** it keeps **track** of an **exponentially decaying average** of **past squared gradients**.

Equation 11-8. Adam algorithm

$$1.\quad \mathbf{m} \leftarrow \beta_1 \mathbf{m} + (1-\beta_1)\nabla_\theta J(\theta)$$

$$2.\quad \mathbf{s} \leftarrow \beta_2 \mathbf{s} + (1-\beta_2)\nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$$

$$3.\quad \mathbf{m} \leftarrow \frac{\mathbf{m}}{1-{\beta_1}^T}$$

$$4.\quad \mathbf{s} \leftarrow \frac{\mathbf{s}}{1-{\beta_2}^T}$$

$$5.\quad \theta \leftarrow \theta - \eta\, \mathbf{m} \oslash \sqrt{\mathbf{s}+\epsilon}$$

- T represents the iteration number (starting at 1).

- If you just look at steps 1, 2, and 5, you will notice Adam's close similarity to both Momentum optimization and RMSProp.
- The only difference is that step 1 computes an exponentially decaying **average** rather than an exponentially decaying **sum**.
- The **momentum decay hyperparameter β₁** is typically initialized to **0.9**, while the **scaling decay hyperparameter β₂** is often initialized to **0.999**.
- TensorFlow's AdamOptimizer class:

```
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
```

- since Adam is an **adaptive learning rate algorithm** (like AdaGrad and RMSProp), it requires **less tuning of the learning rate hyperparameter η**. You can **often use the default value** η = 0.001, **making Adam even easier to use than Gradient Descent**.

3. Explain the techniques required to avoid overfitting through regularization. [July 2023]

1. **Early stopping:** To avoid overfitting the training set, a great solution is early stopping. Just interrupt training when its performance on the validation set starts dropping.
2. **ℓ$_1$ and ℓ$_2$ Regularization:**
   - We can use ℓ1 and ℓ2 regularization to constrain a neural network's connection weights.
   - TensorFlow provides an option to do this. You can pass any function that takes weights as an argument and returns the corresponding regularization loss. The *l1_regularizer()*, *l2_regularizer()*, and *l1_l2_regularizer()* functions return such functions. The following code puts all this together:

```
with arg_scope( [fully_connected],
                weights_regularizer=tf.contrib.layers.l1_regularizer(scale=0.01)
                ):
        hidden1 = fully_connected(X, n_hidden1, scope="hidden1")
        hidden2 = fully_connected(hidden1, n_hidden2, scope="hidden2")
        logits = fully_connected(hidden2, n_outputs, activation_fn=None,scope="out")
```

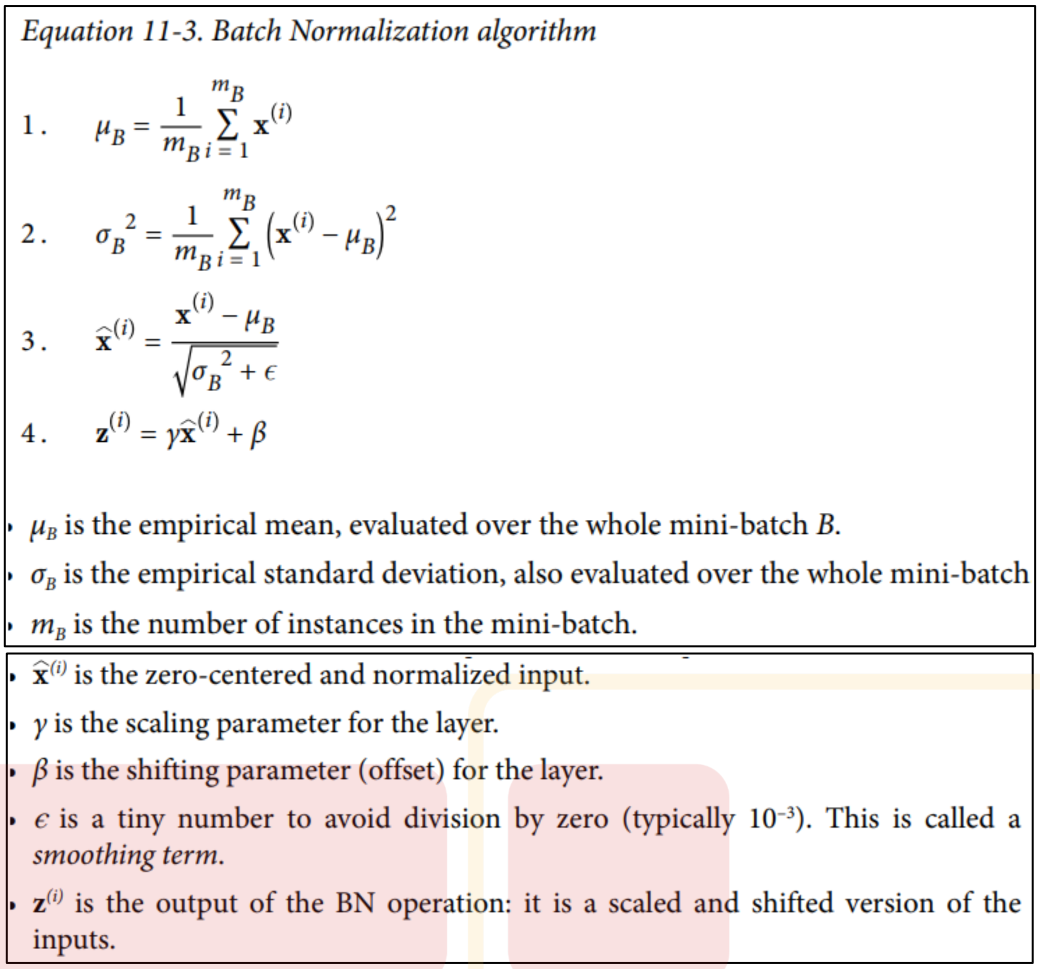
   - This code **creates a neural network with two hidden layers and one output layer**, and it **also creates nodes** in the graph to **compute the ℓ1 regularization loss** corresponding to each layer's weights. **TensorFlow automatically adds these nodes to a special collection containing all the regularization losses**.

- You just need to add these regularization losses to your overall loss.

3. **Dropout:**
    - The **most popular regularization technique** for deep neural networks is arguably dropout.
    - State-of-the-art neural networks got a **1% to 2% accuracy boost** simply **by adding dropout**. This may not sound like a lot, but when a **model already has 95% accuracy**, **getting a 2% accuracy boost means dropping the error rate by almost 40%.** (Going from 5% error to roughly 3%).
    - It is a fairly simple algorithm: **at every training step**, every neuron (including the input neurons but excluding the output neurons) has a probability **p** of being temporarily "**dropped out**," meaning it will be **entirely ignored** during this **training step**, but it may be active during the next step.
    - The hyperparameter **p** is called the **dropout rate**, and it is typically set to **50%.** After training, neurons don't get dropped anymore. And that's all.
    - To implement dropout using **TensorFlow,** you can simply apply the **dropout()** function to the **input layer** and to the **output of every hidden layer**. During **training**, this function **randomly drops some items** (setting them to 0) **and divides the remaining items** by the **keep probability**. After training, this function **does nothing at all**.
4. **Max-Norm Regularization:**
    - Another regularization technique that is quite popular for neural networks is called max-norm regularization: **for each neuron**, it **constrains the weights w** of the incoming connections such that **$\| w \|_2 \le r$**, where **r is the max-norm hyperparameter** and $\| \cdot \|_2$ is the **ℓ_2 norm**.
    - We typically implement this constraint by **computing $\|w\|_2$ after each training step** and **clipping w** if needed.
    - **Reducing r increases** the **amount** of **regularization** and helps **reduce overfitting**.
    - Max-norm regularization can also help alleviate the vanishing/exploding gradients problems.
5. **Data Augmentation:**
    - One last regularization technique, data augmentation, consists of **generating new training instances from existing ones**, artificially **boosting** the **size** of the **training set**.
    - This will **reduce overfitting**, making this a regularization technique.
    - The trick is to **generate realistic training instances**; ideally, **a human should not be able to tell which instances were generated and which ones were not**.
    - Moreover, simply **adding white noise will not help**; the modifications you apply should be learnable.

4. Describe Batch Normalization algorithm with its advantages. [July 2023]

- The technique consists **of adding an operation in the model just before the activation function of each layer**
- Simply **zero-centering and normalizing** the **inputs**, then **scaling** and **shifting** the **result** using **two new parameters per layer** (one for scaling, the other for shifting).
- In other words, this operation lets the **model learn the optimal scale and mean** of the **inputs** for **each layer**.
- In order to zero-center and normalize the inputs, the algorithm **needs to estimate the inputs' mean and standard deviation.**


Equation 11-3. Batch Normalization algorithm

1. $\mu_B = \dfrac{1}{m_B}\sum_{i=1}^{m_B} \mathbf{x}^{(i)}$

2. $\sigma_B{}^2 = \dfrac{1}{m_B}\sum_{i=1}^{m_B} \left(\mathbf{x}^{(i)} - \mu_B\right)^2$

3. $\widehat{\mathbf{x}}^{(i)} = \dfrac{\mathbf{x}^{(i)} - \mu_B}{\sqrt{\sigma_B{}^2 + \epsilon}}$

4. $\mathbf{z}^{(i)} = \gamma\widehat{\mathbf{x}}^{(i)} + \beta$

- μ_B is the empirical mean, evaluated over the whole mini-batch B.
- σ_B is the empirical standard deviation, also evaluated over the whole mini-batch
- m_B is the number of instances in the mini-batch.
- $\widehat{\mathbf{x}}^{(i)}$ is the zero-centered and normalized input.
- γ is the scaling parameter for the layer.
- β is the shifting parameter (offset) for the layer.
- ϵ is a tiny number to avoid division by zero (typically 10^{-3}). This is called a *smoothing term*.
- $\mathbf{z}^{(i)}$ is the output of the BN operation: it is a scaled and shifted version of the inputs.

- **Advantages:**
  - Considerably improved all the deep neural networks.
  - **The vanishing gradients problem was strongly reduced,** to the point that they could use saturating activation functions such as the **tanh** and even the **logistic activation function.**
  - The networks were also **much less sensitive** to the **weight initialization.**
  - They(authors) were able to use **much larger learning rates**, significantly speeding up the learning process.
  - Batch Normalization also acts like a **regularizer, reducing the need for other regularization techniques.**

5. Write a note on i) Xavier and He Initialization ii) Nonsaturating Activation Functions

- **Xavier and He Initialization:**
  - We need the signal to flow properly in both directions: in the forward direction when making predictions, and in the reverse direction when backpropagating gradients.
  - For the signal to flow properly, the authors argue that **we need the variance of the outputs of each layer to be equal to the variance of its inputs**, and we also need the **gradients to have equal variance before and after flowing through a layer in the reverse direction.**
  - It is actually not possible to guarantee both unless the layer has an equal number of input and output connections.
  - but they proposed a **good compromise** that has proven to work very well in practice: the **connection weights must be initialized randomly as described** in **below Equation**, where n_{inputs} and $n_{outputs}$ are the number of input and output connections for the layer whose weights are being initialized.This initialization strategy is often called **Xavier initialization**.

Equation 11-1. Xavier initialization (when using the logistic activation function)

Normal distribution with mean 0 and standard deviation $\sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$

Or a uniform distribution between -r and +r, with $r = \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$

- Using the Xavier initialization strategy can speed up training considerably, and it is one of the tricks that led to the current success of Deep Learning.
- The **initialization strategy for the ReLU activation function** (and its variants, including the ELU activation) is sometimes called **He initialization.**

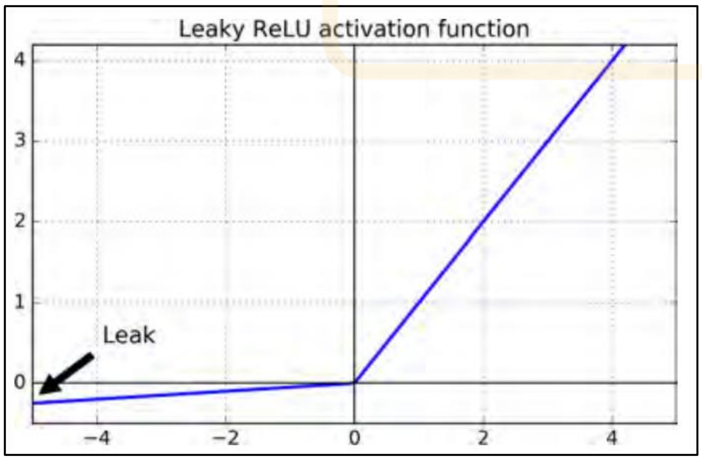
Table 11-1. Initialization parameters for each type of activation function

Activation function	Uniform distribution [–r, r]	Normal distribution
Logistic	$r = \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
Hyperbolic tangent	$r = 4\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = 4\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
ReLU (and its variants)	$r = \sqrt{2}\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{2}\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$

- **Nonsaturating Activation Functions:**
  - The ReLU activation function is **not perfect**. It suffers from a problem known as the **dying ReLUs**: during training, some **neurons effectively die**, meaning they **stop outputting** anything other than 0.
  - In some cases, you may find that **half of your network's neurons are dead**, especially if you **used a large learning rate**.
  - To solve this problem, you may want to use a variant of the ReLU function, such as the **leaky ReLU**. This function is defined as :

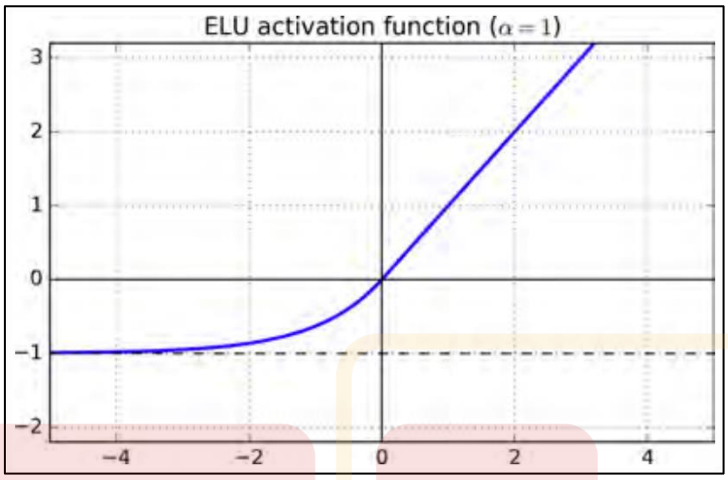
$$\mathbf{LeakyReLU_{\alpha}(z) = \max(\alpha z, z).}$$

  - The **hyperparameter α** defines how much the function "**leaks**": it is the slope of the function **for z < 0, and is typically set to 0.01**. This small slope ensures **that leaky ReLUs never die**; they can go into a long coma, but they have a chance to eventually wake up.



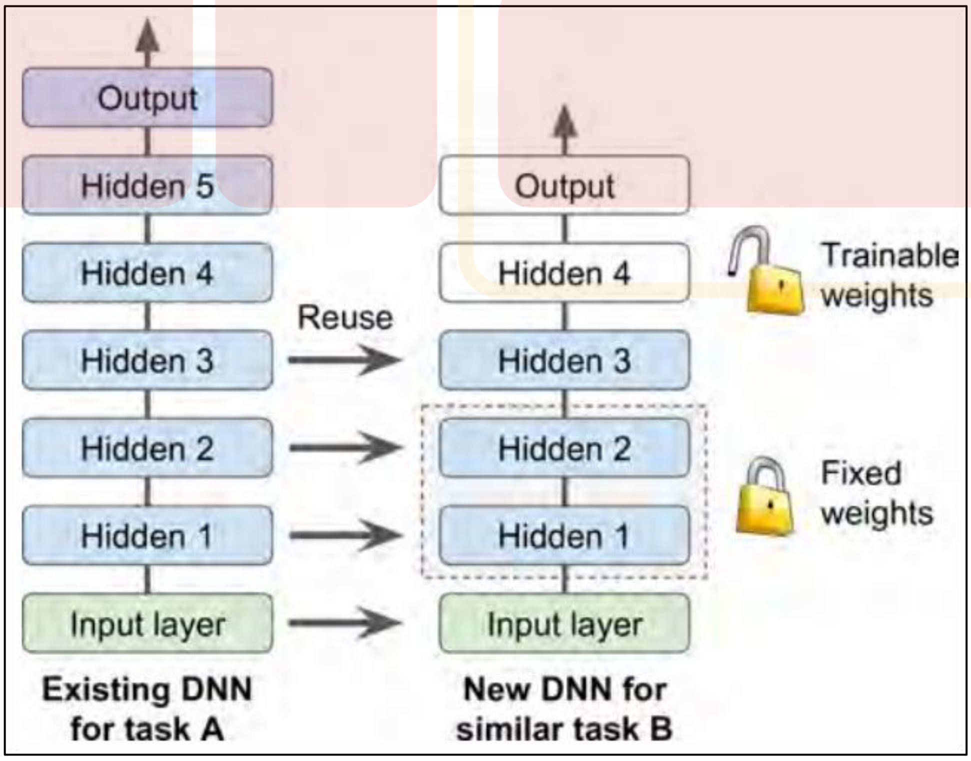
- **Exponential linear unit (ELU)** outperformed all the ReLU variants in their experiments: **training time was reduced** and the neural network performed better on the test set. It is represented as:

$$\mathrm{ELU}_{\alpha}(z) = \begin{cases} \alpha(\exp(z) - 1) & \text{if } z < 0 \\ z & if z \geq 0 \end{cases}$$



6. Write a note on i) Reusing Pretrained Layers ii) Freezing the Lower Layers iii) Caching the Frozen Layers. [July 2023]

- <u>**Reusing Pretrained Layers:**</u>
    - It is generally **not a good idea to train a very large DNN from scratch**.
    - instead, you should always try to find an **existing neural network that accomplishes a similar task** to the one you are trying to tackle, then **just reuse the lower layers of this network:** this is called "**transfer learning**".
    - It will not only **speed up training** considerably, but will also require **much less training data**.



- For example, suppose that you have access to a DNN that was trained to classify pictures into 100 different categories, including animals, plants, vehicles, and everyday objects. You now want to train a DNN to classify specific types of vehicles. These tasks are very similar, so you should try to reuse parts of the first network.

- **Freezing the Lower Layers:**
  - It is likely that **the lower layers of the first DNN** have learned to **detect low-level features** in pictures that will be useful across both image classification tasks, so you can **just reuse these layers as they are.**
  - It is generally a good idea to **"freeze" their weights** when training the new DNN.
  - If the **lower-layer weights are fixed**, then the **higher-layer weights will be easier to train** (because they won't have to learn a moving target).
  - In TensorFlow, to freeze the lower layers during training, the simplest solution is to give the **optimizer the list of variables to train, excluding the variables from the lower layers**:

```
train_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,
                               scope="hidden[34]|outputs")
training_op = optimizer.minimize(loss, var_list=train_vars)
```

  - **The first line gets the list of all trainable variables in hidden layers 3 and 4 and in the output layer.**
  - This **leaves out** the variables in the **hidden layers 1 and 2**.
  - Next we **provide this restricted list** of trainable variables to the **optimizer's minimize()** function. **Layers 1 and 2 are now frozen**: they will not budge during training.

- **Caching the Frozen Layers:**
  - Since the frozen layers won't change, **it is possible to cache the output of the topmost frozen layer** for each training instance.
  - Since training goes through the whole dataset many times, this will give you a **huge speed boost** as you will only need to go through the frozen layers once per training instance.
  - For example, you could first run the **whole training set through the lower layers.**

```
hidden2_outputs = sess.run(hidden2, feed_dict={X: X_train})
```

  - Then during training, instead of building batches of training instances, you would build batches of outputs from hidden layer 2 and feed them to the training operation.

```
import numpy as np

n_epochs = 100
n_batches = 500

for epoch in range(n_epochs):
    shuffled_idx = rnd.permutation(len(hidden2_outputs))
    hidden2_batches = np.array_split(hidden2_outputs[shuffled_idx], n_batches)
    y_batches = np.array_split(y_train[shuffled_idx], n_batches)
    for hidden2_batch, y_batch in zip(hidden2_batches, y_batches):
        sess.run(training_op, feed_dict={hidden2: hidden2_batch, y: y_batch})
```

  - The **last line** runs the training operation defined earlier (which freezes layers 1 and 2), and **feeds it a batch of outputs from the second hidden lay.**
  - Since we give TensorFlow the **output of hidden layer 2, it does not try to evaluate it (or any node it depends on).**

9. What are the advantages over faster optimizer vs regular Gradient Descent optimizer.
(Answer from ChatGPT)

A faster optimizer, often referred to as an adaptive optimizer, such as Adam, RMSprop, or Adagrad, offers several advantages over the regular Gradient Descent optimizer. Here are some of the main advantages:

1. **Adaptive Learning Rates**: Faster optimizers use adaptive learning rates, which adjust the learning rate for each parameter based on past gradients. This helps in efficiently navigating different slopes and speeds up convergence.
2. **Faster Convergence**: These optimizers often converge more quickly than regular Gradient Descent because they adapt the learning rate during training. This can save training time and computational resources.
3. **Robustness**: Adaptive optimizers are generally more robust to different types of data and problems because they adjust the learning rate on a per-parameter basis. This can lead to better performance on noisy or sparse datasets.
4. **Avoiding Local Minima**: The adaptive learning rates help the optimizer navigate the loss landscape more effectively, potentially avoiding local minima and finding a better global minimum.
5. **No Need for Extensive Hyperparameter Tuning**: Regular Gradient Descent requires careful tuning of the learning rate, which can be a time-consuming process. Adaptive optimizers typically require less manual tuning of hyperparameters, as they adjust the learning rates automatically.
6. **Momentum and Scaling**: Some adaptive optimizers, such as Adam, include momentum and scaling terms that help accelerate the convergence process and smooth out the optimization path.
7. **Easier to Handle Sparse Gradients**: For problems where gradients are sparse (i.e., most gradients are zero), adaptive optimizers can be more effective because they adjust learning rates based on the magnitude of past gradients.

While these advantages make adaptive optimizers appealing, it's important to note that they may not always perform better in every situation. Depending on the problem, regular Gradient Descent or other optimization methods might still be the best choice.

10. Explain the different techniques used in Faster Optimizers. (The answer is more elaborated, just in case they are asked as individual questions for Sem)

1. <u>**Momentum Optimization:**</u>
    - Momentum optimization cares about what **previous gradients** were.
    - At **each iteration**, it **adds** the **local gradient** to the **momentum vector m**, and it **updates the weights** by simply subtracting this momentum vector.

Equation 11-4. Momentum algorithm

$$1.\quad \mathbf{m} \leftarrow \beta\mathbf{m} + \eta\nabla_\theta J(\theta)$$

$$2.\quad \theta \leftarrow \theta - \mathbf{m}$$

- To simulate some sort of friction mechanism **and prevent the momentum from growing too large**, the algorithm introduces a new **hyperparameter β**, simply called the momentum, which must be **set between 0 (high friction) and 1 (no friction**).
- A typical momentum value **is 0.9**.

```
optimizer = tf.train.MomentumOptimizer(learning_rate=learning_rate,
                                       momentum=0.9)
```

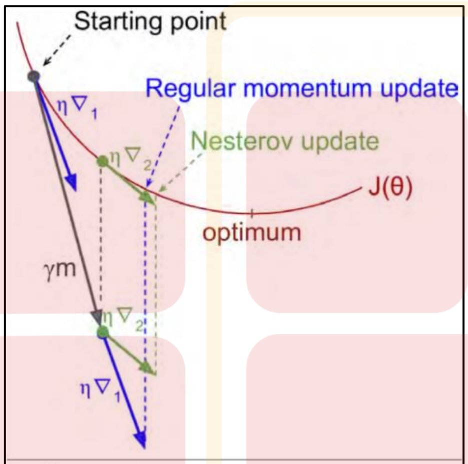
2. **Nesterov Accelerated Gradient:**
   - One small variant to Momentum optimization, is almost always faster than vanilla Momentum optimization.
   - The idea of Nesterov Momentum optimization, or Nesterov Accelerated Gradient (NAG), is to **measure the gradient of the cost function not at the local position but slightly ahead in the direction of the momentum**.

Equation 11-5. Nesterov Accelerated Gradient algorithm

$$1.\quad \mathbf{m} \leftarrow \beta\mathbf{m} + \eta\nabla_\theta J(\theta + \beta\mathbf{m})$$

$$2.\quad \theta \leftarrow \theta - \mathbf{m}$$

   - The only difference from vanilla Momentum optimization is that the **gradient is measured at θ + βm rather than at θ.**



Regular versus Nesterov Momentum optimization

```
optimizer = tf.train.MomentumOptimizer(learning_rate=learning_rate,
                                       momentum=0.9, use_nesterov=True)
```

3. **AdaGrad:**
   - Gradient Descent starts by quickly going down the steepest slope, then slowly goes down the bottom of the valley. The **AdaGrad algorithm** can **detect this early on** and **correct** its **direction to point** a **bit more toward the global optimum**.
   - AdaGrad achieves this by **scaling down the gradient vector along** the **steepest dimensions**.

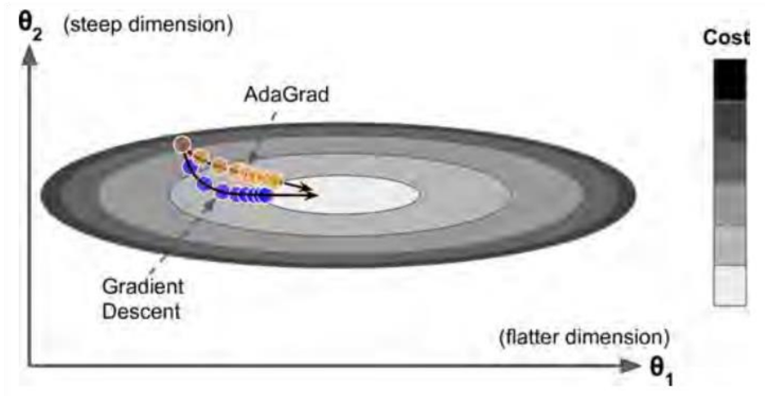
Equation 11-6. AdaGrad algorithm

$$1.\quad \mathbf{s} \leftarrow \mathbf{s} + \nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$$

$$2.\quad \theta \leftarrow \theta - \eta\,\nabla_\theta J(\theta) \oslash \sqrt{\mathbf{s} + \epsilon}$$

   - If the **cost function is steep** along the **i**th **dimension**, then s_i will get **larger and larger** at each **iteration**.
   - This algorithm **decays the learning rate**, but it does so **faster for steep dimensions** than for dimensions with gentler slopes. This is called an **adaptive learning rate**. It helps point the resulting updates **more directly toward the global optimum**.

- One additional benefit is that it requires **much less tuning** of the **learning rate hyperparameter η**.



AdaGrad versus Gradient Descent

4. **RMSProp:**
   - **AdaGrad** algorithm **slows** down a **bit too fast** and ends up **never converging** to the **global optimum**.
   - The **RMSProp** algorithm **fixes** this by **accumulating only** the **gradients** from the **most recent iterations** (as opposed to all the gradients since the beginning of training).
   - It does so by **using exponential decay** in the **first step**:

Equation 11-7. RMSProp algorithm

$$1.\quad \mathbf{s} \leftarrow \beta \mathbf{s} + (1-\beta)\nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$$

$$2.\quad \theta \leftarrow \theta - \eta \, \nabla_\theta J(\theta) \oslash \sqrt{\mathbf{s} + \epsilon}$$

   - The **decay rate β** is typically set to **0.9**.
   - **β** is a new hyperparameter, but this default value often works well, so you may not need to tune it at all.
   - TensorFlow has an RMSPropOptimizer class:

```
optimizer = tf.train.RMSPropOptimizer(learning_rate=learning_rate,
                                      momentum=0.9, decay=0.9, epsilon=1e-10)
```

   - This optimizer almost **always performs much better than AdaGrad**. It was the preferred optimization algorithm of many researchers **until Adam optimization came around**.
5. <u>**Adam Optimization:**</u>
   - Adam, which stands for **adaptive moment estimation**, **combines** the **ideas** of **Momentum optimization** and **RMSProp**.
   - Just like **Momentum optimization** it keeps **track** of an **exponentially decaying average** of **past gradients**, and just like **RMSProp** it keeps **track** of an **exponentially decaying average** of **past squared gradients**.

Equation 11-8. Adam algorithm

$$\begin{aligned}
&1. \quad \mathbf{m} \leftarrow \beta_1 \mathbf{m} + (1-\beta_1)\nabla_\theta J(\theta)\\
&2. \quad \mathbf{s} \leftarrow \beta_2 \mathbf{s} + (1-\beta_2)\nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)\\
&3. \quad \mathbf{m} \leftarrow \frac{\mathbf{m}}{1-{\beta_1}^T}\\
&4. \quad \mathbf{s} \leftarrow \frac{\mathbf{s}}{1-{\beta_2}^T}\\
&5. \quad \theta \leftarrow \theta - \eta\,\mathbf{m} \oslash \sqrt{\mathbf{s}+\epsilon}
\end{aligned}$$

- T represents the iteration number (starting at 1).

- If you just look at steps 1, 2, and 5, you will notice Adam's close similarity to both Momentum optimization and RMSProp.
- The only difference is that step 1 computes an exponentially decaying **average** rather than an exponentially decaying **sum**.
- The **momentum decay hyperparameter β₁** is typically initialized to **0.9**, while the **scaling decay hyperparameter β₂** is often initialized to **0.999**.
- TensorFlow's AdamOptimizer class:

```
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
```

- since Adam is an **adaptive learning rate algorithm** (like AdaGrad and RMSProp), it requires **less tuning of the learning rate hyperparameter η**. You can **often use the default value** η = 0.001, **making Adam even easier to use than Gradient Descent**.

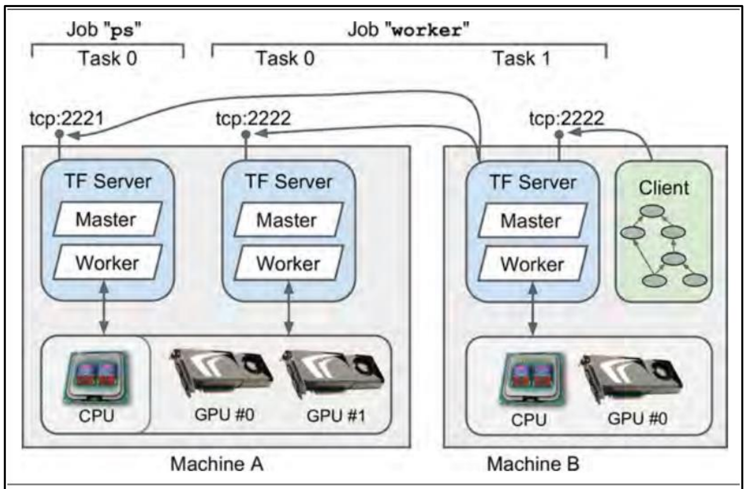
MODULE 3: Distributing Tensor flow across devices and servers | Convolution Neural Network

1. What is the difference between pinning an operation on a device and placing an operation on a device? Explain the different strategies used to placing operation on a device.

-------xx------

2. Explain with a neat diagram how a graph on multiple devices is executed across multiple servers.

- To run a graph across multiple servers, you first need to **define a cluster**.
- A **cluster** is composed of **one or more TensorFlow servers**, called **tasks**, typically spread across several machines as shown above. **Each task belongs to a job**.
- A **job** is just a **named group of tasks** that typically have a **common role**, such as keeping track of the **model parameters** (Job "ps") or **performing computation** (Job "worker")**.**



12-6. *TensorFlow cluster*

- In this example**, machine A hosts two TensorFlow servers** (i.e., tasks), **listening on different ports: one is, part of the "ps" job, and the other is part of the "worker" job.**
- **Machine B just hosts one TensorFlow server, part of the "worker" job**.

```
cluster_spec = tf.train.ClusterSpec({
    "ps": [
        "machine-a.example.com:2221",  # /job:ps/task:0
    ],
    "worker": [
        "machine-a.example.com:2222",  # /job:worker/task:0
        "machine-b.example.com:2222",  # /job:worker/task:1
    ]})
```

- **To start a TensorFlow server**, you must create a **Server object**, passing it the **cluster specification,** and its **own job name** and **task number**.
- For example, **to start the first worker task,** you would run the following code **on machine A**:

```
server = tf.train.Server(cluster_spec, job_name="worker", task_index=0)
```

- If you want the process to do **nothing** other than **run** the TensorFlow **server**, you can **block the main thread** by telling it to **wait for the server** to finish; **using the join() method** :

```
server.join()  # blocks until the server stops (i.e., never)
```

3. Write a note on the Master and Worker Services in Tensorflow.

- **Every TensorFlow server** provides **two services**: the master service and the worker service.
- The **master** service **allows clients** to **open sessions** and use them to **run graphs.**
- It **coordinates** the **computations across tasks**, relying on the **worker service** to **actually execute computations** on **other tasks** and get their **results**.
- This architecture gives you a lot of **flexibility**.
- **One client** can **connect** to **multiple servers** by **opening multiple sessions** in **different threads**.
- **One server** can **handle multiple sessions simultaneously** from **one or more clients.**
- You can **run one client per task** (typically within the same process), or **just one client to control all tasks**.

4. Write a note on i) Sharding Variables Across Multiple Parameter Servers. ii) Sharing State Across Sessions Using Resource Containers. [July 2023]

1. **Sharding Variables Across Multiple Parameter Servers:**
    - For **large models** with **millions of parameters**, it is useful to **shard these parameters** across **multiple parameter servers**, to **reduce the risk of saturating a single parameter server's network card.**
    - TensorFlow provides the **replica_device_setter()** function, which distributes variables across **all the "ps" tasks** in a **round-robin fashion**.
    - For example, the following code **pins five variables** to **two parameter servers**:

```
with tf.device(tf.train.replica_device_setter(ps_tasks=2):
    v1 = tf.Variable(1.0)  # pinned to /job:ps/task:0
    v2 = tf.Variable(2.0)  # pinned to /job:ps/task:1
    v3 = tf.Variable(3.0)  # pinned to /job:ps/task:0
    v4 = tf.Variable(4.0)  # pinned to /job:ps/task:1
    v5 = tf.Variable(5.0)  # pinned to /job:ps/task:0
```

    - Instead of passing the number of **ps_tasks**, you can pass the **cluster spec** cluster=cluster_spec and TensorFlow will simply **count the number of tasks in the "ps" job**.
2. **Sharing State Across Sessions Using Resource Containers:**
    - When you are using a **plain local session** (not the distributed) each **variable's state** is **managed by the session itself**; as soon as it **ends**, **all variable** values are **lost.**
    - **Multiple local sessions cannot share any state**, even if they both run the same graph; each session has its own copy of every variable.
    - **A better option is to use a container block.**

```
with tf.container("my_problem_1"):
    [...] # Construction phase of problem 1
```

    - One advantage is that variable names remain nice and short. Another advantage is that you can easily reset a named container.
    - Resource containers make it **easy to share variables across sessions in flexible ways**.
    - Below diagram shows **four clients** running **different graphs** on the **same cluster**, but **sharing some variables.**

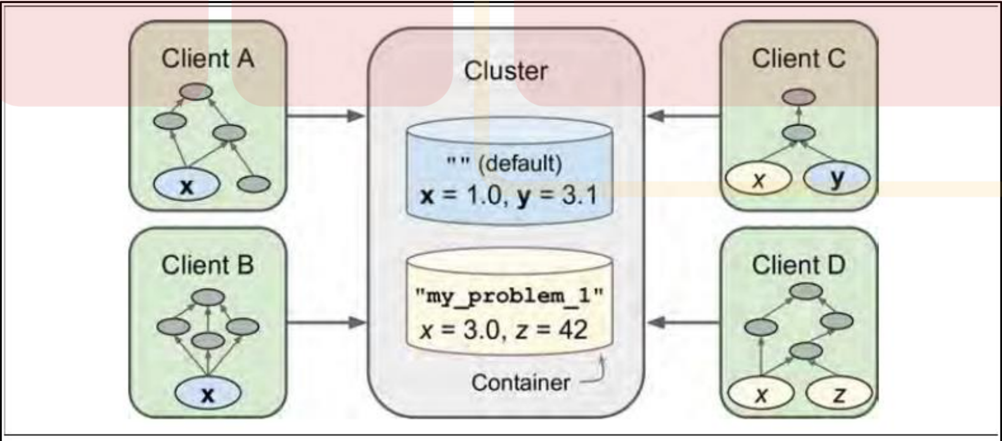


Figure 12-7. Resource containers

5. Describe how asynchronous Communication is achieved Using TensorFlow Queues

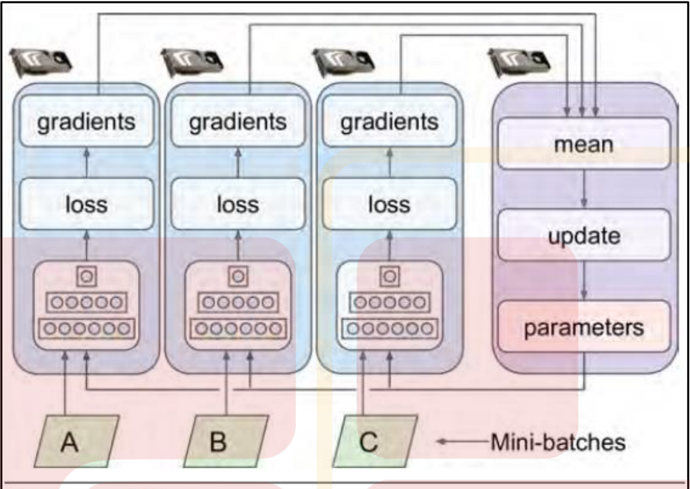
-----

6. Describe the two major approaches to handling a neural network ensemble. Also write the pros and cons of these methods.

-----

7. Define Data parallelism. Briefly explain Synchronous updates, Asynchronous updates and Bandwidth Saturation. [July 2023]

- Another way to **parallelize the training** of a neural network is to **replicate it on each device**, **run** a training step **simultaneously** on **all replicas using** a **different mini-batch for each**, and then **aggregate the gradients** to **update the model parameters**. This is called "**data parallelism".**



. *Data parallelism*

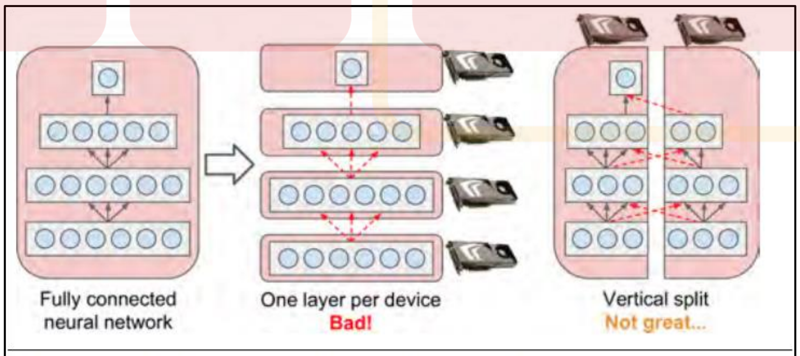
- There are two variants of this approach: synchronous updates and asynchronous updates.
- <u>**Synchronous updates:**</u>
    - With synchronous updates, the **aggregator waits for all gradients** to be **available** before **computing the average** and **applying the result**.
    - Once a **replica** has **finished computing** its gradients, it **must wait** for the **parameters** to be **updated** before it can **proceed** to the **next minibatch**.
    - The **downside** is that **some devices** may be **slower** than **others**, so **all other devices** will have to **wait** for **them** at **every step**.
    - Moreover, the **parameters will be copied** to **every device** almost at the **same time**, which may **saturate the parameter servers' bandwidth**.
- <u>**Asynchronous updates:**</u>
    - With asynchronous updates, **whenever** a **replica** has **finished computing** the **gradients**, it **immediately uses** them to **update** the model **parameters**.
    - There is **no aggregation**, and **no synchronization**.
    - Replicas just work **independently** of the other replicas.
    - Since there is **no waiting** for the other replicas, this approach **runs more training steps per minute.**
    - Moreover, **risk of bandwidth saturation is reduced.**
    - By the time a replica has finished computing the gradients based on some parameter values, **these parameters will have been updated several times by other replicas** and there is **no guarantee** that the **computed gradients** will still be **pointing in the right direction**.

- **When gradients are severely out-of-date**, they are called **stale gradients**: they can **slow down convergence**, introducing **noise** and **wobble effects** or they can even make the training algorithm **diverge**.
- There are a few ways to **reduce** the effect of **stale gradients**:
  - **Reduce the learning rate**.
  - **Drop** stale gradients or **scale them down**.
  - **Adjust** the **mini-batch size**.
  - **Start** the **first few epochs** using **just one replica**

- <u>**Bandwidth saturation:**</u>
  - Whether you use synchronous or asynchronous updates, **data parallelism still requires communicating** the model parameters **from the parameter servers** to **every replica** at the **beginning** of every training step, and the **gradients** in the **other direction** at the **end of each** training step.
  - Unfortunately, this means that there always comes a point where **adding an extra GPU** will **not improve performance at all**.
  - At that point, adding more GPUs will just **increase saturation and slow down training**.
  - Saturation is **more severe for large dense models**, since they have **a lot of parameters** and **gradients** to **transfer**.
  - Here are a few simple steps you can take to **reduce the saturation problem**:
    - **Group your GPUs on a few servers** rather than **scattering them across many servers**. This will **avoid unnecessary network hops**.
    - **Shard the parameters across multiple parameter servers**.
    - **Drop** the **model parameters' float precision** from **32 bits (tf.float32)** to **16 bits (tf.bfloat16)**. This will **cut in half the amount of data to transfer**, **without much impact** on the **convergence rate** or the **model's performance**.

8. Define Model parallelism. Explain the different approaches used in Model parallelism.

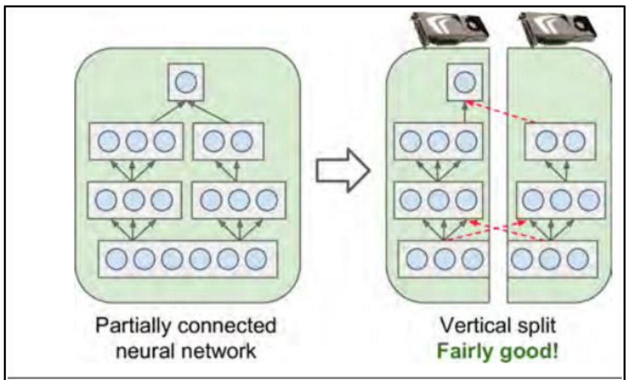
- If we want to **run a single** neural network across **multiple devices**, this requires **chopping your model into separate chunks** and **running each chunk on a different device**. This is called **model parallelism.**
- Model parallelism depends on the **architecture of your neural network.**
- For fully connected networks, there is generally not much to be gained from this approach.



. *Splitting a fully connected neural network*

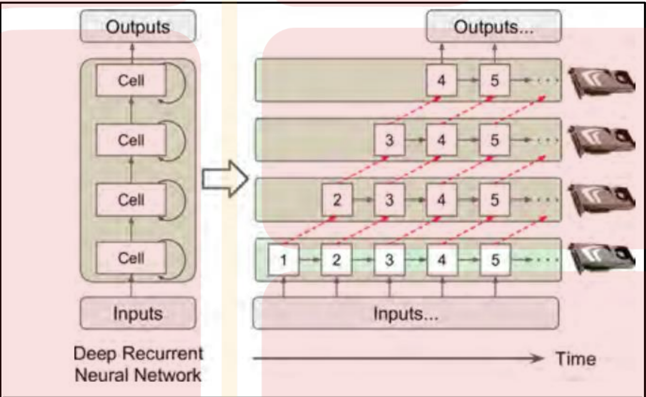
  - it may seem that an **easy way to split the model is to place each layer on a different device**, but **this does not work** since **each layer** needs to **wait** for the **output of the previous layer** before it can do anything.
  - you can slice it **vertically**, but the problem is that **each half of the next layer requires the output of both halves**, so there will be a **lot of cross-device communication.**

    - This is likely to **completely cancel out the benefit** of the **parallel** computation, since cross-device communication is **slow**.
- Some neural network architectures, such as **convolutional neural networks**, contain **layers** that are **only partially connected to the lower layers**, so it is much **easier to distribute chunks across devices** in an **efficient** way.



Splitting a partially connected neural network

- Some **deep recurrent neural networks** are composed of **several layers of memory cells.**
    - If you split such a network **horizontally,** placing each layer on a different device, then at the **first step: only one device will be active,** at the **second step: two will be active**, and **by the time** the **signal** propagates to the **output** layer **all devices will be active simultaneously**.
    - There is still a **lot** of **cross-device communication** going on, but since **each cell may be fairly complex,** the **benefit** of running multiple cells in parallel often **outweighs** the **communication penalty.**



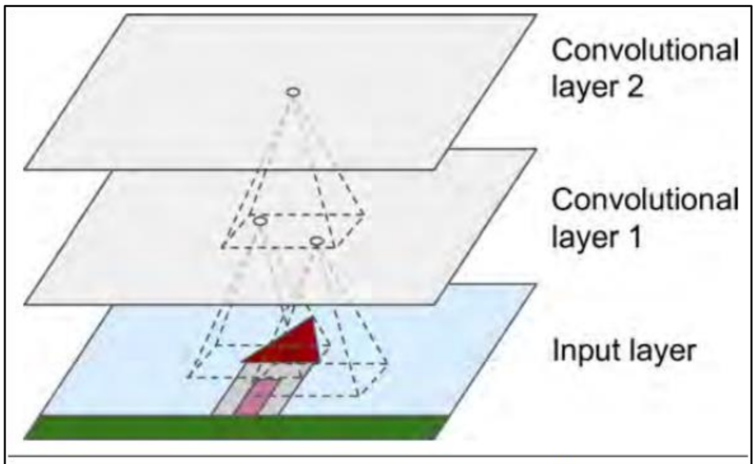
Splitting a deep recurrent neural network

9. Explain Convolution Neural Network with the Architecture of the Visual Cortex.

------

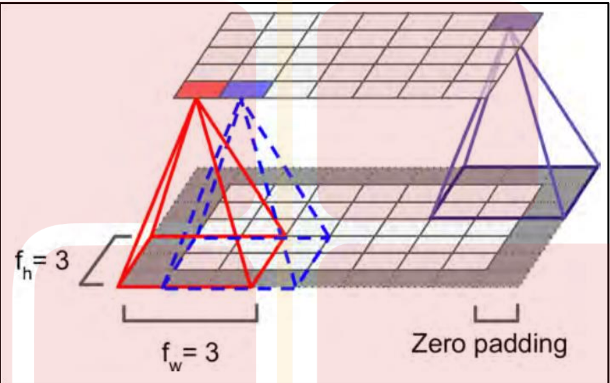
10. Write a note on Convolutional Layer.

- The most important building block of a CNN is the convolutional layer.
- Neurons in the first convolutional layer are **not** connected to every single pixel in the input image, but only to **pixels in their receptive fields**.
- In turn, each neuron in the **second** convolutional layer is connected **only to neurons located within** a **small rectangle in the first layer.**
- This architecture allows the **network to concentrate on low-level features** in the **first hidden layer**, **then assemble** them into **higher-level features in the next hidden layer**, and **so on**.
- This **hierarchical** structure is **common** in **real-world images,** which is one of the reasons why CNNs **work so well for image recognition**.



CNN layers with rectangular local receptive fields

- A neuron located in **row i, column j** of a **given layer** is **connected** to the **outputs** of the **neurons** in the **previous layer** located in **rows i to i + f_h – 1**, **columns j to j + f_w – 1**, where **f_h and f_w** are the **height and width of the receptive field.**
- In order for **a layer to have the same height and width** as **the previous layer**, it is common to **add zeros around the inputs**, as shown below. This is called **zero padding.**



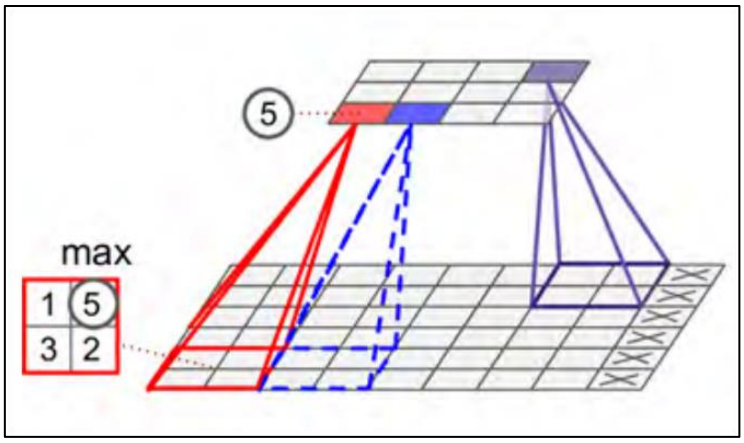
Connections between layers and zero padding

11.Explain Fully convolutional network and discuss how you can convert a dense layer into a Convolutional Layer. [July 2023]

----

12. Describe pooling layer in Convolutional Layer.

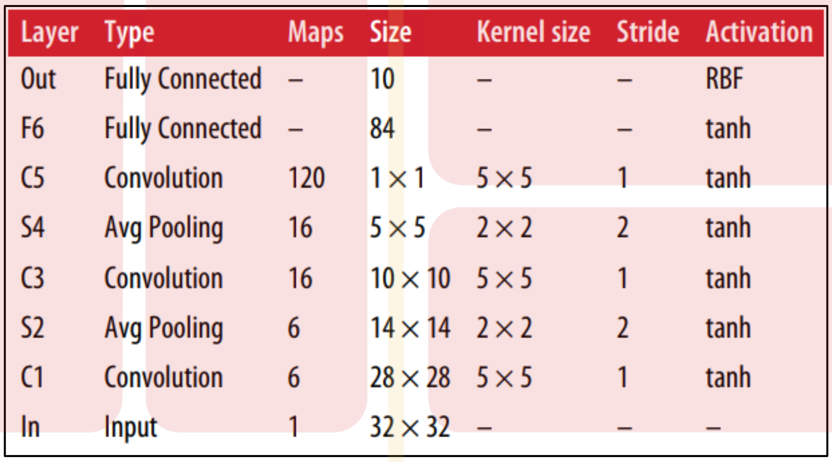
- The goal of a Pooling Layer is to **subsample** (i.e., shrink) the **input image** in order to **reduce** the **computational load, the memory usage, and the number of parameters** (thereby **limiting** the **risk** of **overfitting**).
- **Reducing the input image size** also makes the **neural network tolerate** a **little bit of image shift.**
- Each **neuron** in a **pooling** layer is **connected to the outputs of a limited number of neurons in the previous layer**, located within a **small rectangular receptive field**.
- You must **define its size**, the **stride**, and the **padding type** just like ordinary convolutional layers.
- **However, a pooling neuron has no weights**; all it does is **aggregate the inputs using an aggregation function such as the max or mean.**
- Below diagram shows a **max pooling layer**, which is the most common type of pooling layer.



- In this example, we use a **2 × 2 pooling kernel**, a **stride of 2**, and **no padding**. Note that only the **max input value in each kernel makes it to the next layer**. The other inputs are **dropped**.
- A pooling layer typically works on **every input channel independently**, so the o**utput depth is the same as the input depth**.
- You may alternatively **pool over the depth dimension**, in which case the i**mage's spatial dimensions (height and width) remain unchanged**, but the **number of channels is reduced**.

13. Compare and Contrast Alex Net architecture with LeNet-5 architecture. [July 2023]

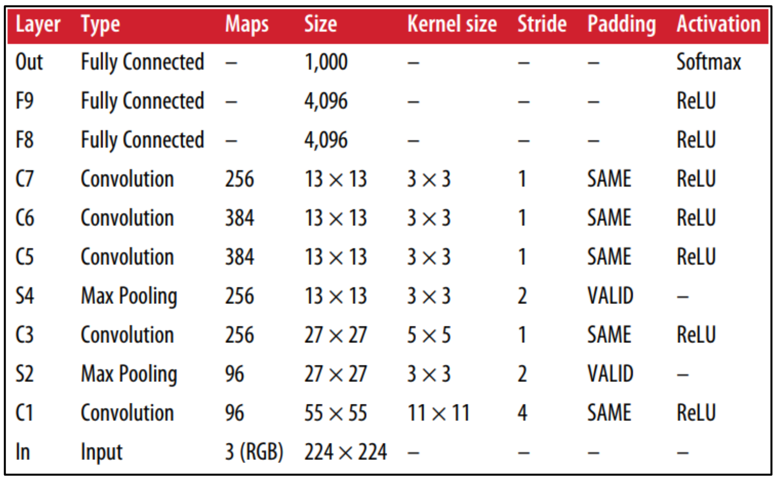
1. **LeNet-5:**
   - The LeNet-5 architecture is perhaps the **most widely known** CNN architecture.
   - widely used for **hand-written digit recognition** (**MNIST**). It is composed of the layers shown below:



Layer	Type	Maps	Size	Kernel size	Stride	Activation
Out	Fully Connected	–	10	–	–	RBF
F6	Fully Connected	–	84	–	–	tanh
C5	Convolution	120	1 × 1	5 × 5	1	tanh
S4	Avg Pooling	16	5 × 5	2 × 2	2	tanh
C3	Convolution	16	10 × 10	5 × 5	1	tanh
S2	Avg Pooling	6	14 × 14	2 × 2	2	tanh
C1	Convolution	6	28 × 28	5 × 5	1	tanh
In	Input	1	32 × 32	–	–	–

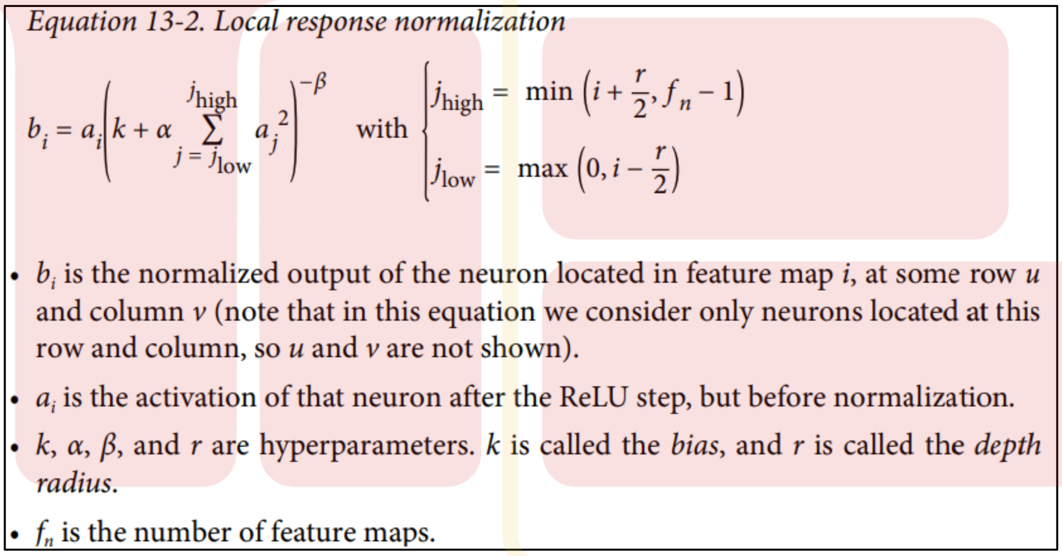
   - **There are a few extra details to be noted:**
     i. MNIST images are **28 × 28 pixels, but they are zero-padded to 32 × 32 pixels** and **normalized** before being fed to the network.
     ii. The rest of the network **does not use any padding**, which is why the **size** keeps **shrinking** as the **image progresses** through the network.
     iii. The **average pooling layers** are slightly **more complex** than usual.
     iv. The output layer; **instead** of **computing the dot product** of the **inputs** and the **weight** vector, **each neuron outputs the square of the Euclidian distance** between its **input vector and its weight vector**.
2. **Alex Net:**
   - The AlexNet CNN architecture **won the 2012 ImageNet ILSVRC challenge** by **a large margin**: it achieved **17% top-5 error rate** while **the second best achieved only 26%.**

- It is **quite similar to LeNet-5**, only **much larger and deeper**.
- It was the **first to stack convolutional layers directly on top of each other**, instead of **stacking a pooling layer on top of each convolutional layer**.



Layer	Type	Maps	Size	Kernel size	Stride	Padding	Activation
Out	Fully Connected	–	1,000	–	–	–	Softmax
F9	Fully Connected	–	4,096	–	–	–	ReLU
F8	Fully Connected	–	4,096	–	–	–	ReLU
C7	Convolution	256	13×13	3×3	1	SAME	ReLU
C6	Convolution	384	13×13	3×3	1	SAME	ReLU
C5	Convolution	384	13×13	3×3	1	SAME	ReLU
S4	Max Pooling	256	13×13	3×3	2	VALID	–
C3	Convolution	256	27×27	5×5	1	SAME	ReLU
S2	Max Pooling	96	27×27	3×3	2	VALID	–
C1	Convolution	96	55×55	11×11	4	SAME	ReLU
In	Input	3 (RGB)	224×224	–	–	–	–

- **To reduce overfitting, the authors used two regularization techniques:**
  i. First, they applied **dropout** (with a **50% dropout rate**) to the **outputs** of layers **F8** and **F9** during **training**.
  ii. Second, they performed **data augmentation** by **randomly shifting** the **training** images by various **offsets**, **flipping** them **horizontally**, and **changing** the **lighting** conditions.
- AlexNet also uses a **competitive normalization** step **immediately after** the **ReLU** step of **layers C1 and C3**, called **local response normalization**.



Equation 13-2. Local response normalization

$$b_i = a_i \left(k + \alpha \sum_{j = j_{\text{low}}}^{j_{\text{high}}} {a_j}^2 \right)^{-\beta} \quad \text{with} \begin{cases} j_{\text{high}} = \min\left(i + \frac{r}{2}, f_n - 1\right) \\ j_{\text{low}} = \max\left(0, i - \frac{r}{2}\right) \end{cases}$$

- b_i is the normalized output of the neuron located in feature map i, at some row u and column v (note that in this equation we consider only neurons located at this row and column, so u and v are not shown).
- a_i is the activation of that neuron after the ReLU step, but before normalization.
- k, α, β, and r are hyperparameters. k is called the *bias*, and r is called the *depth radius*.
- f_n is the number of feature maps.

MODULE 4: Recurrent Neural Network

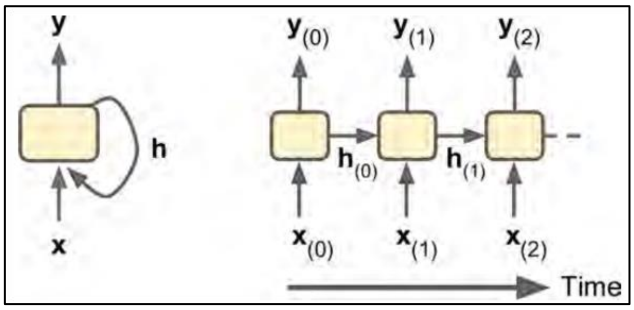
1. With a neat diagram, explain the following: i) Memory Cell ii) Input Output Sequence in RNN iii) Peephole Connections [July 2023]

1. Memory Cell:

- Since the **output** of a **recurrent neuron** at time step **t** is a function of **all the inputs** from **previous time steps**, you could say it has a form of **memory**.
- A part of a neural network that **preserves some state across time steps** is called a **memory cell** (or simply a cell).
- In general, a cell's state at time step **t**, denoted **$h_{(t)}$** ("h" stands for "hidden"), is a **function** of some **inputs** at that **time step** and its **state** at the **previous time step**:

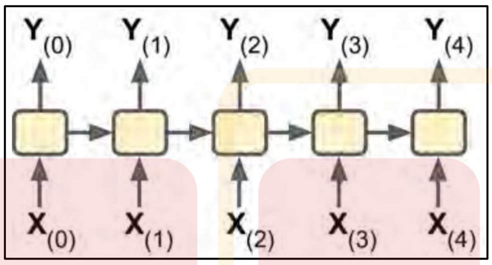
$$h_{(t)} = f(h_{(t-1)}, x_{(t)}).$$

- Its output at time step **t**, denoted $\mathbf{y_{(t)}}$, is **also a function** of the **previous state** and the **current inputs**.

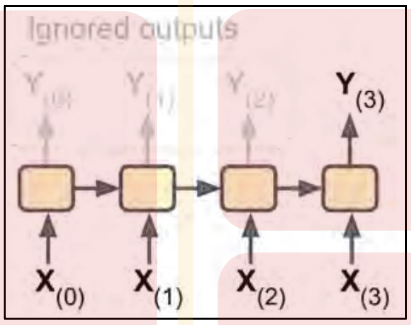


2. Input Output Sequence:

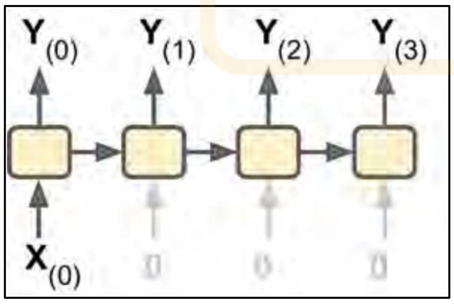
- An RNN can **simultaneously** take a **sequence** of **inputs** and produce a **sequence** of **outputs.** This is called **Sequence-to-Sequence** network**. For example**, this type of network is useful for predicting time series such as stock prices.



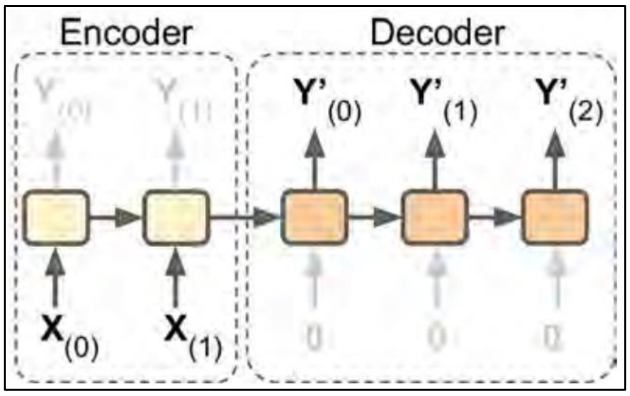
- Alternatively, you could feed the network a **sequence of inputs**, and **ignore all outputs except for the last one.** This is a **Sequence-to-Vector** network. **For example**, this type of network is useful to output a sentiment score from movie review.



- Conversely, you could feed the network **a single input** at the **first time step** (and zeros for all other time steps), and let it **output a sequence**. This is a **Vector-to-Sequence** network**. For example**, this type of network is useful when the input is an image, and the output could be a caption for that image.



- Lastly, you could have a **Sequence-to-Vector** network, called an **encoder**, followed by a **Vector-to-Sequence** network, called a **decoder**. **For example**, this can be used for translating a sentence from one language to another.



3. Peephole Connections:

- In a basic **LSTM** cell, the **gate controllers** can look only at the input $\mathbf{x}_{(t)}$ and the previous **short-term** state $\mathbf{h}_{(t-1)}$
- It may be a good idea to give them a **bit more context** by letting them **peek at the long-term state** as well.
- This led to an LSTM variant with **extra connections** called **peephole connections**: the **previous long-term** state $\mathbf{c}_{(t-1)}$ is added as an **input** to the **controllers** of the **forget gate** and the **input gate**, and the **current long-term state** $c_{(t)}$ is added as **input** to the **controller** of the **output gate**.
- To implement peephole connections in TensorFlow, you must use the **LSTMCell** instead of the **BasicLSTMCell** and set **use_peepholes=True**

```
lstm_cell = tf.contrib.rnn.LSTMCell(num_units=n_neurons, use_peepholes=True)
```

2. What are the advantages of building a RNN using dynamic_rnn() rather than static_rnn() method. [July 2023]

- The **static_rnn()** function creates an **unrolled RNN** network by chaining cells.
- This approach **builds a graph** containing **one cell per time step**. If there were **50 time steps**, the **graph would look pretty ugly**.
- It is a bit like writing a program **without** ever using **loops.**
- With such a **large graph**, you may even get **out-of-memory (OOM) errors** during **backpropagation** (especially with the limited memory of GPU cards), since it must **store all tensor values** during the **forward pass** so it can **use them** to compute **gradients** during the **reverse pass.**
- the **dynamic_rnn()** function is a better solution.
- The dynamic_rnn() function uses a **while_loop() operation** to run over the cell **appropriate number** of times.
- You can set **swap_memory=True** if you want it to **swap the GPU's memory** to the **CPU's memory** during backpropagation to **avoid OOM errors**.
- Conveniently, it also **accepts a single tensor for all inputs at every time step** and it **outputs** a **single tensor** for **all outputs at every time step**.
- There is **no need to stack, unstack, or transpose**.
- The following code creates the RNN using the dynamic_rnn() function:

```
X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])

basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
outputs, states = tf.nn.dynamic_rnn(basic_cell, X, dtype=tf.float32)
```

3. Describe the techniques required to distribute training and execution of deep RNN across multiple GPUs. [July 2023]

- We can **efficiently distribute** deep RNNs across **multiple GPUs** by **pinning each layer** to a **different GPU.**
- However, if you try to **create each cell** in a **different device() block**, it **will not work.** This fails because a **BasicRNNCell** is a **cell factory,** not a **cell.** No cells get created when you create the factory, and thus no variables either. The device block is simply **ignored**.
- The **trick** is to **create your own cell wrapper**:

```
import tensorflow as tf

class DeviceCellWrapper(tf.contrib.rnn.RNNCell):
  def __init__(self, device, cell):
    self._cell = cell
    self._device = device

  @property
  def state_size(self):
    return self._cell.state_size

  @property
  def output_size(self):
    return self._cell.output_size

  def __call__(self, inputs, state, scope=None):
    with tf.device(self._device):
        return self._cell(inputs, state, scope)
```

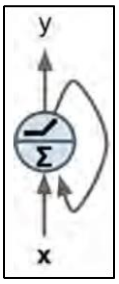
- This wrapper simply **proxies** every **method call** to **another cell,** except it **wraps** the **__call__()** function **within a device block**.
- **Now you can distribute each layer on a different GPU:**

```
devices = ["/gpu:0", "/gpu:1", "/gpu:2"]
cells = [DeviceCellWrapper(dev,tf.contrib.rnn.BasicRNNCell(num_units=n_neurons))
         for dev in devices]
multi_layer_cell = tf.contrib.rnn.MultiRNNCell(cells)
outputs, states = tf.nn.dynamic_rnn(multi_layer_cell, X, dtype=tf.float32)
```

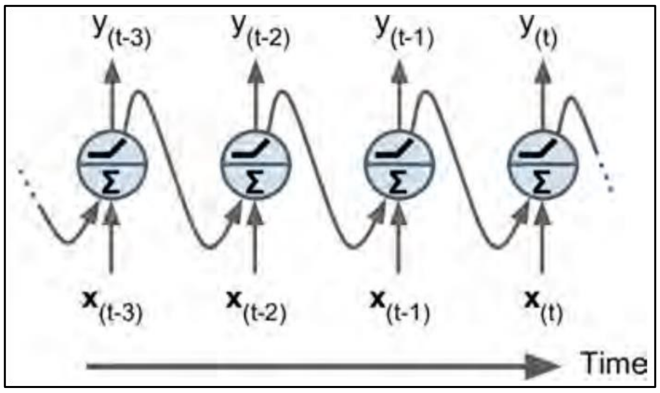
- **Do not** set **state_is_tuple=False**, or the MultiRNNCell will **concatenate all the cell states** into a **single tensor,** on a **single GPU.**

4. Give the structure of recurrent neurons and explain the working

- A recurrent neural network looks very much like a **feedforward neural network**, **except** it also has **connections pointing backward**.
- the simplest possible RNN, composed of just **one neuron receiving inputs**, **producing an output**, and **sending that output back to itself**, as shown below:



- At each **time step t** (also called a **frame**), this recurrent neuron **receives the inputs $\mathbf{x}_{(t)}$** as well as its **own output** from **the previous time step**, $\mathbf{y}_{(t-1)}$
- We can represent this tiny network against the **time axis**, as shown below. This is called **unrolling the network through time**.



- Each recurrent neuron **has two sets of weights**: one for the inputs $\mathbf{x}_{(t)}$ and the other for the outputs of the previous time step, $\mathbf{y}_{(t-1)}$. Let's call these weight vectors $\mathbf{w}_x$ and $\mathbf{w}_y$.
- The **output** of a **single recurrent neuron** can be computed as:

$$\boxed{\mathbf{y}_{(t)} = \phi\left({\mathbf{x}_{(t)}}^T \cdot \mathbf{w}_x + {\mathbf{y}_{(t-1)}}^T \cdot \mathbf{w}_y + b\right)}$$

- we can compute a whole layer's output in one shot for a whole mini-batch using a vectorized form of the previous equation:

$$\begin{aligned} \mathbf{Y}_{(t)} &= \phi\left(\mathbf{X}_{(t)} \cdot \mathbf{W}_x + \mathbf{Y}_{(t-1)} \cdot \mathbf{W}_y + \mathbf{b}\right) \\ &= \phi\left(\left[\mathbf{X}_{(t)} \quad \mathbf{Y}_{(t-1)}\right] \cdot \mathbf{W} + \mathbf{b}\right) \text{ with } \mathbf{W} = \begin{bmatrix} \mathbf{W}_x \\ \mathbf{W}_y \end{bmatrix} \end{aligned}$$

- $\mathbf{Y}_{(t)}$ is an $m \times n_{\text{neurons}}$ matrix containing the layer's outputs at time step t for each instance in the mini-batch (m is the number of instances in the mini-batch and n_{neurons} is the number of neurons).
- $\mathbf{X}_{(t)}$ is an $m \times n_{\text{inputs}}$ matrix containing the inputs for all instances (n_{inputs} is the number of input features).
- $\mathbf{W}_x$ is an $n_{\text{inputs}} \times n_{\text{neurons}}$ matrix containing the connection weights for the inputs of the current time step.
- $\mathbf{W}_y$ is an $n_{\text{neurons}} \times n_{\text{neurons}}$ matrix containing the connection weights for the outputs of the previous time step.
- The weight matrices $\mathbf{W}_x$ and $\mathbf{W}_y$ are often concatenated into a single weight matrix $\mathbf{W}$ of shape $(n_{\text{inputs}} + n_{\text{neurons}}) \times n_{\text{neurons}}$ (see the second line of Equation 14-2).
- $\mathbf{b}$ is a vector of size n_{neurons} containing each neuron's bias term.

5. Describe how RNNs are used in Training to Predict Time Series

- We will train an **RNN** to **predict** the **next value** in a generated time series.

- **Each** training **instance** is a **randomly** selected sequence of **20 consecutive values** from the time series, and the **target sequence** is the **same** as the **input sequence**, except it is **shifted by one time step into the future.**
- Let's create the RNN with **100** recurrent neurons and **20** time steps.

```
n_steps = 20
n_inputs = 1
n_neurons = 100
n_outputs = 1

X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
y = tf.placeholder(tf.float32, [None, n_steps, n_outputs])
```

- **Wrap** the **BasicRNNCell** into an **OutputProjectionWrapper**

```
cell = tf.contrib.rnn.OutputProjectionWrapper(
    tf.contrib.rnn.BasicRNNCell(num_units=n_neurons, activation=tf.nn.relu),
    output_size=n_outputs)
```

- Now we need to **define** the **cost function**. We will use the **Mean Squared Error (MSE).**
- Next, we will create an **Adam optimizer**, the **training op**, and the **variable initialization op**.

```
learning_rate = 0.001

loss = tf.reduce_mean(tf.square(outputs - y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(loss)

init = tf.global_variables_initializer()
```

- Now the execution phase:

```
n_iterations = 10000
batch_size = 50

with tf.Session() as sess:
    init.run()
    for iteration in range(n_iterations):
        X_batch, y_batch = [...]  # fetch the next training batch
        sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
        if iteration % 100 == 0:
            mse = loss.eval(feed_dict={X: X_batch, y: y_batch})
            print(iteration, "\tMSE:", mse)
```

- The program's **output** should look like this:

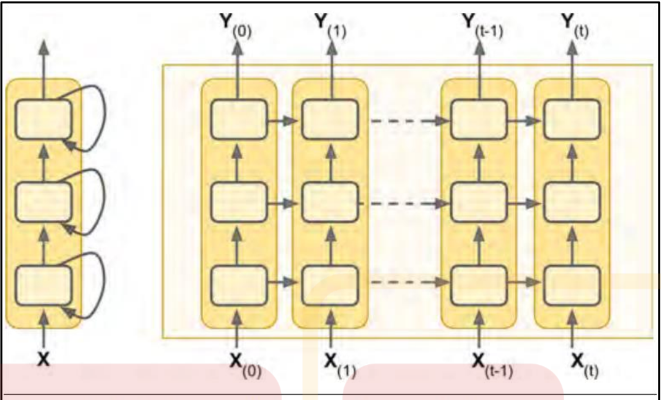
```
0    MSE: 379.586
100  MSE: 14.58426
200  MSE: 7.14066
300  MSE: 3.98528
400  MSE: 2.00254
[...]
```

- Once the model is trained, you can make predictions.

```
X_new = [...]  # New sequences
y_pred = sess.run(outputs, feed_dict={X: X_new})
```

6. Write a note on the following i) Deep RNNs ii) GRU Cell

1. **Deep RNNs:**
   - It is quite common to **stack multiple layers of cells**, This gives you a **deep RNN**.



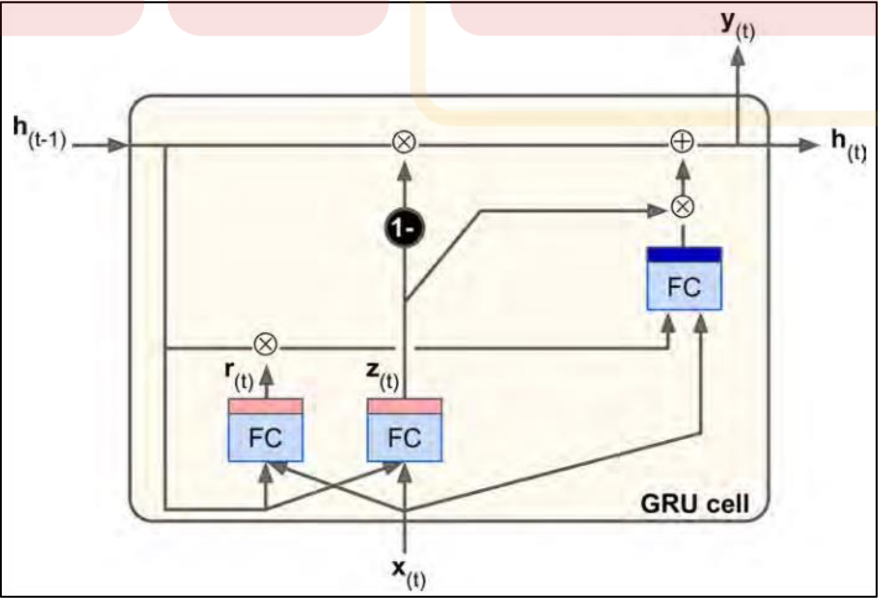
Deep RNN (left), unrolled through time (right)

   - To implement a **deep RNN in TensorFlow**, you can **create several cells** and **stack** them into a **MultiRNNCell**. In the following code we stack three identical cells:

```
n_neurons = 100
n_layers = 3

basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
multi_layer_cell = tf.contrib.rnn.MultiRNNCell([basic_cell] * n_layers)
outputs, states = tf.nn.dynamic_rnn(multi_layer_cell, X, dtype=tf.float32)
```

   - The **states** variable is a **tuple** containing **one tensor per layer**, each representing the **final state of that layer's cell.**
   - If you set **state_is_tuple=False** when creating the **MultiRNNCell**, then states becomes a **single tensor containing the states from every layer**, concatenated along the column axis.

2. **GRU Cell:**
   - The GRU cell is a **simplified version** of the **LSTM** cell, and it seems to perform just as well.



- The main simplifications are:
    i. Both state vectors are merged into a single vector **$h_{(t)}$**
    ii. A **single gate controller controls** both the **forget gate and the input gate**. If the **gate** controller outputs **1**, the **input gate is open** and the **forget gate is closed**. If it outputs **0**, the **opposite** happens.
    iii. There is **no output gate**; the **full state vector is output at every time step**.
- Below **equation summarizes** how to **compute** the cell's **state at each time step.**

Equation 14-4. GRU computations

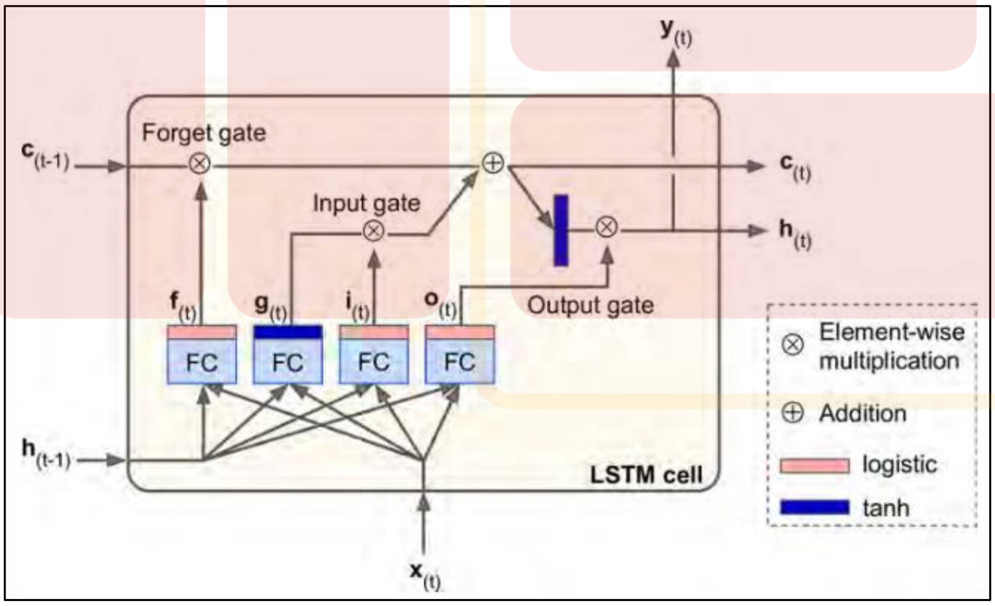
$$\begin{aligned}
\mathbf{z}_{(t)} &= \sigma\left(\mathbf{W}_{xz}{}^{T}\cdot\mathbf{x}_{(t)} + \mathbf{W}_{hz}{}^{T}\cdot\mathbf{h}_{(t-1)}\right)\\
\mathbf{r}_{(t)} &= \sigma\left(\mathbf{W}_{xr}{}^{T}\cdot\mathbf{x}_{(t)} + \mathbf{W}_{hr}{}^{T}\cdot\mathbf{h}_{(t-1)}\right)\\
\mathbf{g}_{(t)} &= \tanh\left(\mathbf{W}_{xg}{}^{T}\cdot\mathbf{x}_{(t)} + \mathbf{W}_{hg}{}^{T}\cdot\left(\mathbf{r}_{(t)}\otimes\mathbf{h}_{(t-1)}\right)\right)\\
\mathbf{h}_{(t)} &= \left(1-\mathbf{z}_{(t)}\right)\otimes\tanh\left(\mathbf{W}_{xg}{}^{T}\cdot\mathbf{h}_{(t-1)} + \mathbf{z}_{(t)}\otimes\mathbf{g}_{t}\right)
\end{aligned}$$

- Creating a GRU cell in **TensorFlow:**

```
gru_cell = tf.contrib.rnn.GRUCell(num_units=n_neurons)
```

7. Give the architecture of a basic LSTM cell and explain its working.

- The **Long Short-Term Memory (LSTM)** can be used very much like a **basic cell**, except it will **perform much better**; **training converge is faster** and it will **detect long-term dependencies** in the data.
- The architecture of a basic LSTM cell is shown below:



- The main layer is the one that outputs **$g_{(t)}$** .It has the usual role of **analyzing** the **current inputs $x_{(t)}$** and the **previous (short-term) state $h_{(t-1)}$** .In an LSTM cell **this layer's output** does **not go straight out,** but instead it **is partially stored in the long-term state**.
- The **three other layers are gate controllers**. Since they use the **logistic activation** function, their outputs range from 0 to 1. If they **output 0s**, they **close** the **gate**, and if they **output 1s**, they **open** it. Specifically:

- The **forget gate** (controlled by **f(t)**) controls, **which parts** of the long-term state **should** be **erased**.
- The **input gate** (controlled by **i(t)**) controls, **which parts** of **g(t)** should be **added** to the long-term state.
- The **output gate** (controlled by **o(t)**) controls, **which parts** of the long-term state should be **read and output at this time step**.

- In short, an LSTM cell can learn to recognize an important input, store it in the long-term state, learn to preserve it for as long as it is needed, and learn to extract it whenever it is needed.
- Below **equation summarizes** how to compute the cell's **long-term state, its short-term state, and its output** at each time step for a single instance:

Equation 14-3. LSTM computations

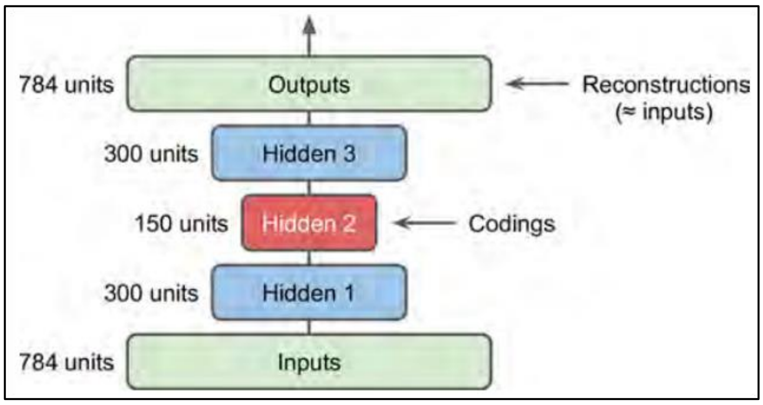
$$
\begin{aligned}
\mathbf{i}_{(t)} &= \sigma\left(\mathbf{W}_{xi}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hi}{}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_{i}\right)\\
\mathbf{f}_{(t)} &= \sigma\left(\mathbf{W}_{xf}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hf}{}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_{f}\right)\\
\mathbf{o}_{(t)} &= \sigma\left(\mathbf{W}_{xo}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{ho}{}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_{o}\right)\\
\mathbf{g}_{(t)} &= \tanh\left(\mathbf{W}_{xg}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hg}{}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_{g}\right)\\
\mathbf{c}_{(t)} &= \mathbf{f}_{(t)} \otimes \mathbf{c}_{(t-1)} + \mathbf{i}_{(t)} \otimes \mathbf{g}_{(t)}\\
\mathbf{y}_{(t)} &= \mathbf{h}_{(t)} = \mathbf{o}_{(t)} \otimes \tanh\left(\mathbf{c}_{(t)}\right)
\end{aligned}
$$

8. Describe the process of combining a convolutional neural network with RNN to classify the videos. [July 2023]

MODULE 5: Autoencoders | Reinforcement Learning

1. With a neat diagram code snippet explain stacked autoencoders.

- autoencoders can have **multiple hidden layers**. In this case they are called **stacked autoencoders** (or deep autoencoders).
- Adding more layers helps the autoencoder **learn more complex codings**.
- The architecture of a stacked autoencoder is **typically symmetrical** with regards to the **central hidden layer** (the coding layer).
- For example, an autoencoder for **MNIST** may have **784 inputs**, followed by a **hidden layer with 300** neurons, then a **central hidden layer of 150 neurons**, then **another hidden layer with 300 neurons**, and an **output layer with 784 neurons**. This stacked autoencoder is represented as:



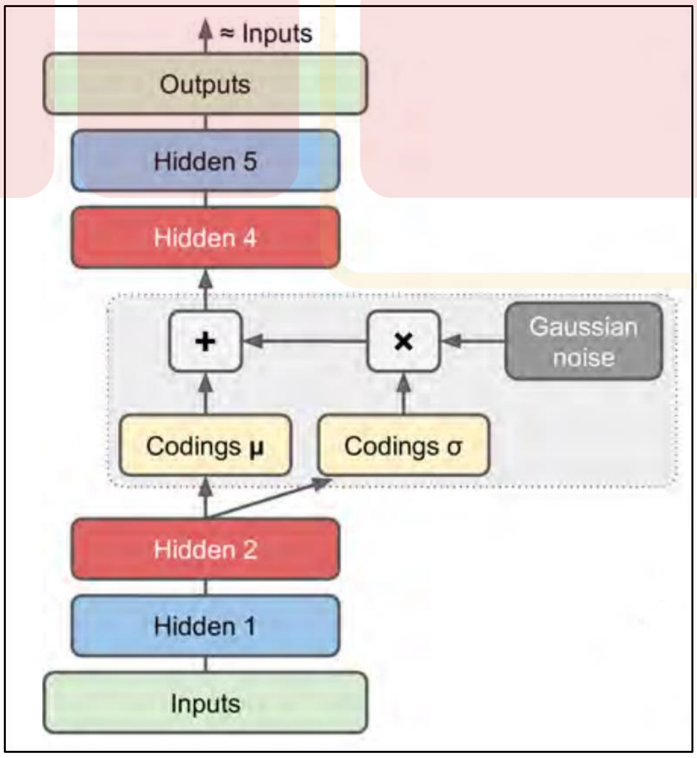
- The following code snippet builds a stacked autoencoder for MNIST, using He initialization, the ELU activation function, and ℓ2 regularization:

```
n_inputs = 28 * 28  # for MNIST
n_hidden1 = 300
n_hidden2 = 150  # codings
n_hidden3 = n_hidden1
n_outputs = n_inputs
```

```
with tf.contrib.framework.arg_scope(
        [fully_connected],
        activation_fn=tf.nn.elu,
        weights_initializer=tf.contrib.layers.variance_scaling_initializer(),
        weights_regularizer=tf.contrib.layers.l2_regularizer(l2_reg)):
    hidden1 = fully_connected(X, n_hidden1)
    hidden2 = fully_connected(hidden1, n_hidden2)  # codings
    hidden3 = fully_connected(hidden2, n_hidden3)
    outputs = fully_connected(hidden3, n_outputs, activation_fn=None)

reconstruction_loss = tf.reduce_mean(tf.square(outputs - X))  # MSE

reg_losses = tf.get_collection(tf.GraphKeys.REGULARIZATION_LOSSES)
loss = tf.add_n([reconstruction_loss] + reg_losses)

optimizer = tf.train.AdamOptimizer(learning_rate)
training_op = optimizer.minimize(loss)

init = tf.global_variables_initializer()
```

2. With a neat diagram code snippet explain variational autoencoders required for generating fashion MNIST images. [July 2023]

- **Variational autoencoders** are quite different:
  - They are **probabilistic autoencoders**, meaning that their **outputs are partly determined** by **chance**, even **after training**.
  - Most importantly, they are **generative autoencoders**, meaning that they can **generate new instances** that **look like they were sampled from the training set**.



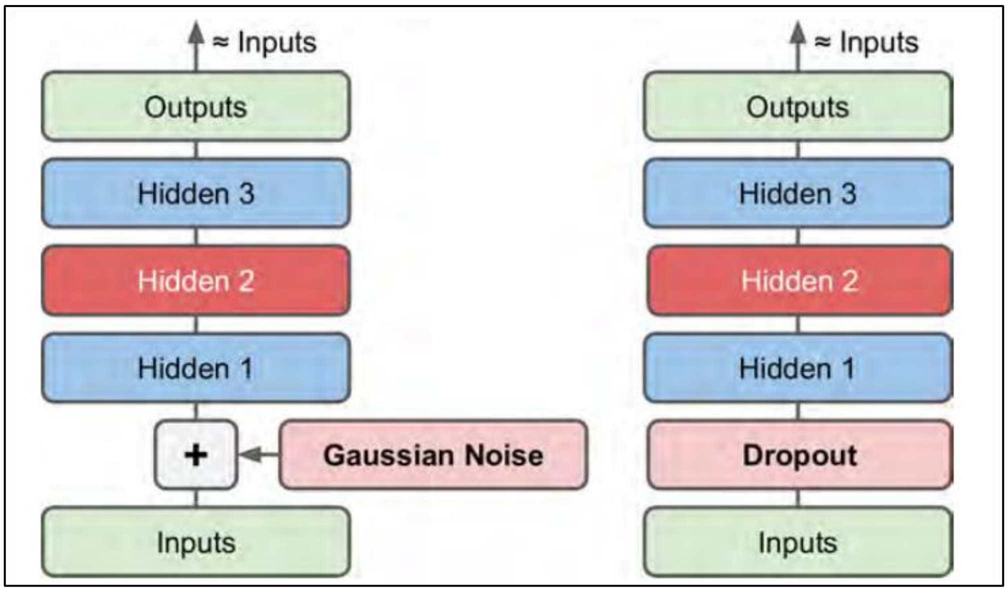
- The following code builds the variational autoencoder shown in diagram above for generating fashion MNIST images:

```
n_inputs = 28 * 28  # for MNIST
n_hidden1 = 500
n_hidden2 = 500
n_hidden3 = 20  # codings
n_hidden4 = n_hidden2
n_hidden5 = n_hidden1
n_outputs = n_inputs

learning_rate = 0.001
```

```
with tf.contrib.framework.arg_scope(
        [fully_connected],
        activation_fn=tf.nn.elu,
        weights_initializer=tf.contrib.layers.variance_scaling_initializer()):
    X = tf.placeholder(tf.float32, [None, n_inputs])


    hidden1 = fully_connected(X, n_hidden1)
    hidden2 = fully_connected(hidden1, n_hidden2)
    hidden3_mean = fully_connected(hidden2, n_hidden3, activation_fn=None)
    hidden3_gamma = fully_connected(hidden2, n_hidden3, activation_fn=None)
    hidden3_sigma = tf.exp(0.5 * hidden3_gamma)
    noise = tf.random_normal(tf.shape(hidden3_sigma), dtype=tf.float32)
    hidden3 = hidden3_mean + hidden3_sigma * noise
    hidden4 = fully_connected(hidden3, n_hidden4)
    hidden5 = fully_connected(hidden4, n_hidden5)
    logits = fully_connected(hidden5, n_outputs, activation_fn=None)
    outputs = tf.sigmoid(logits)

reconstruction_loss = tf.reduce_sum(
    tf.nn.sigmoid_cross_entropy_with_logits(labels=X, logits=logits))
latent_loss = 0.5 * tf.reduce_sum(
    tf.exp(hidden3_gamma) + tf.square(hidden3_mean) - 1 - hidden3_gamma)
cost = reconstruction_loss + latent_loss

optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(cost)

init = tf.global_variables_initializer()
```

3. Describe Denoising Autoencoders with its architecture and implementation.

- Another way to **force** the autoencoder to **learn useful features** is to **add noise** to its **inputs**, **training** it to **recover** the **original**, noise-free inputs.
- This **prevents** the autoencoder from **trivially copying its inputs to its outputs**, so it ends up having to **find patterns in the data**.
- The noise can be **pure Gaussian noise** added to the inputs, or it can be **randomly switched off inputs**, just like in **dropout.**



- The Tensorflow implementation:
  - Implementing denoising autoencoders in TensorFlow is not too hard. Let's start with **Gaussian noise**:

```
X = tf.placeholder(tf.float32, shape=[None, n_inputs])
X_noisy = X + tf.random_normal(tf.shape(X))
[...]
hidden1 = activation(tf.matmul(X_noisy, weights1) + biases1)
[...]
reconstruction_loss = tf.reduce_mean(tf.square(outputs - X))  # MSE
[...]
```

  - Implementing the **dropout version**, which is more common, is not much harder:

```
from tensorflow.contrib.layers import dropout

keep_prob = 0.7

is_training = tf.placeholder_with_default(False, shape=(), name='is_training')
X = tf.placeholder(tf.float32, shape=[None, n_inputs])
X_drop = dropout(X, keep_prob, is_training=is_training)
[...]
hidden1 = activation(tf.matmul(X_drop, weights1) + biases1)
[...]
reconstruction_loss = tf.reduce_mean(tf.square(outputs - X))  # MSE
[...]
```

  - During training we must set **is_training** to **True** using the **feed_dict** :

```
sess.run(training_op, feed_dict={X: X_batch, is_training: True})
```

4.What is OpenAI Gym? Explain the working of cartpole environment using OpenAI Gym. [July 2023]

-

5. Explain Reinforcement learning? Determine the features of reinforcement learning by comparing with regular supervised and unsupervised learning. [July 2023]

-

6. Using evaluation action of Reinforcement learning, propose a solution for credit assignment problem and justify how can you improve the rewards assigned. [July 2023]

-