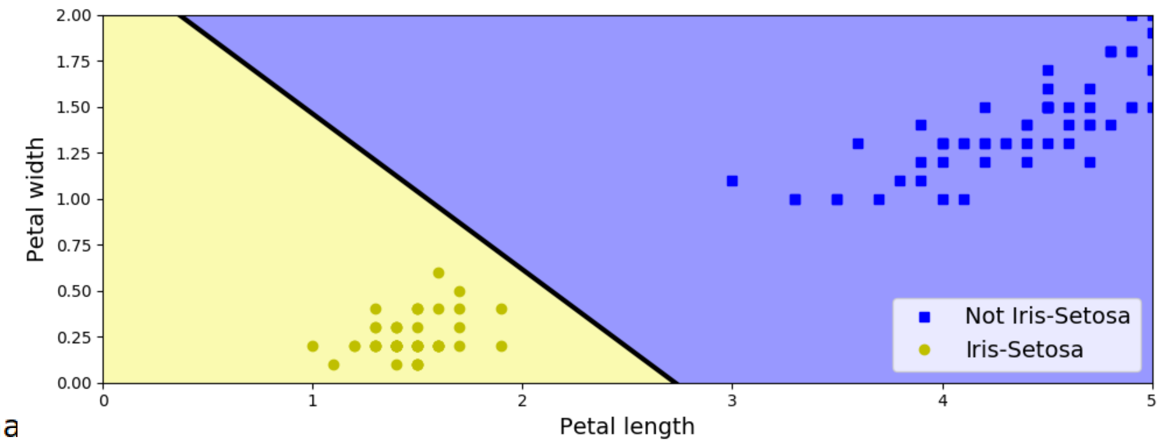
Artficial Neural Network Internals
By Mohit Kumar

Artificial Neural Network:Perceptron

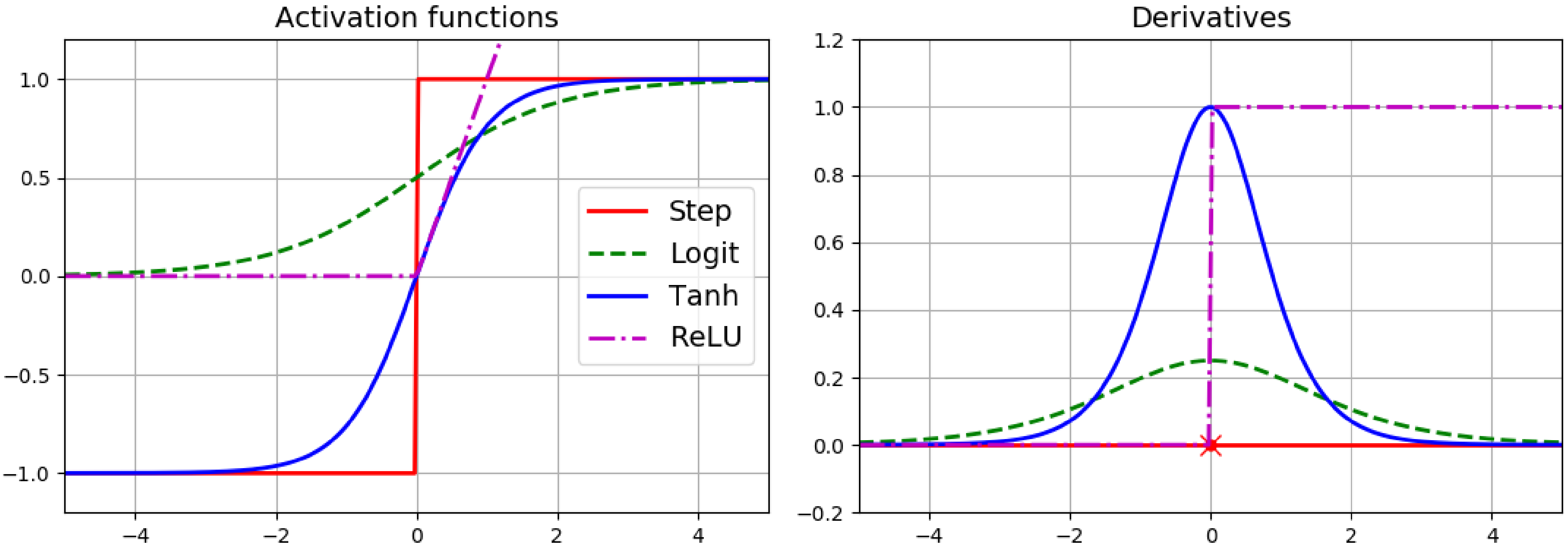
```
from sklearn.datasets import load_iris
import numpy as np
import numpy.random as rnd
from sklearn.linear_model import Perceptron

iris = load_iris()
X = iris.data[:, (2, 3)]  # petal length, petal width
y = (iris.target == 0).astype(np.int)


per_clf = Perceptron(random_state=42)
per_clf.fit(X, y)

y_pred = per_clf.predict([[2, 0.5]])
print("y_pred:",y_pred)
```

Artificial Neural Network:Perceptron:ContourPlot

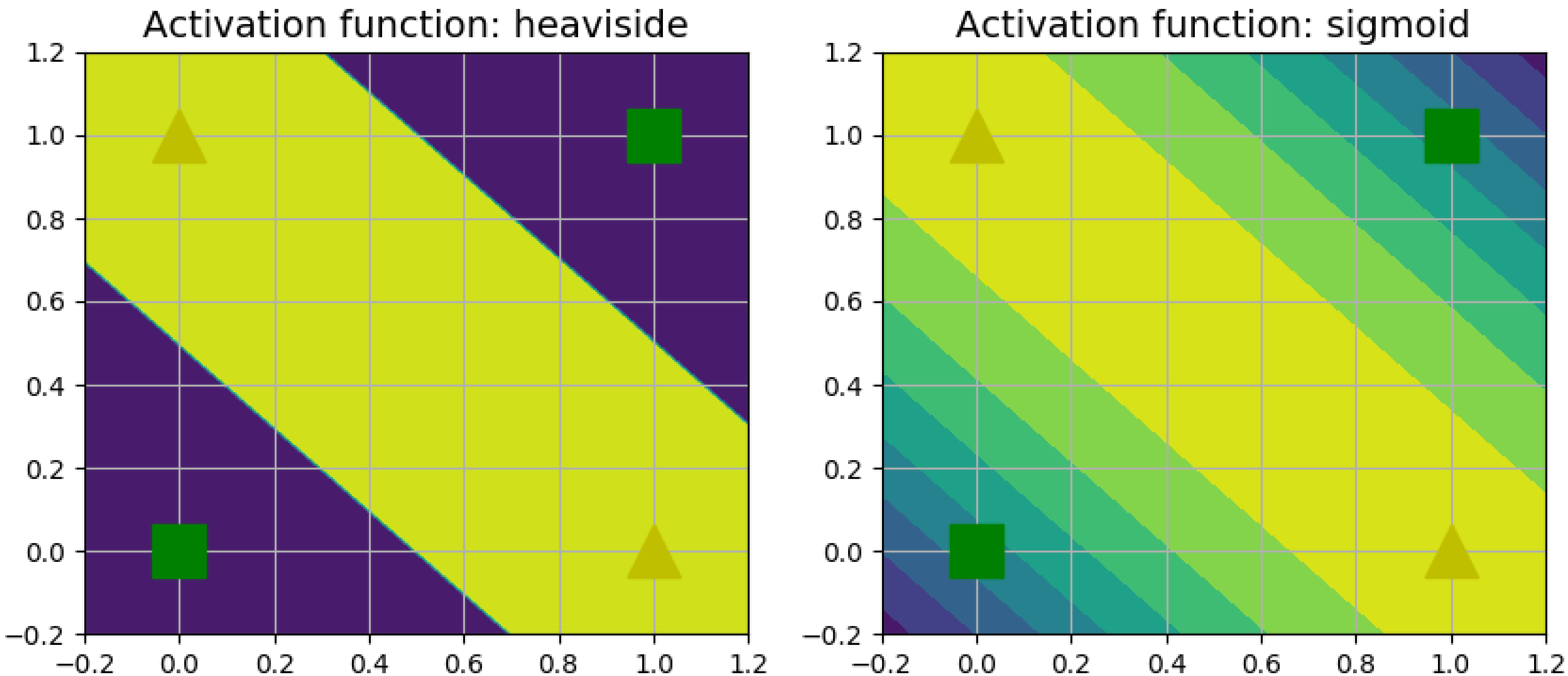
```
iris = load_iris()
X = iris.data[:, (2, 3)]  # petal length, petal width
y = (iris.target == 0).astype(np.int)


per_clf = Perceptron(random_state=42)
per_clf.fit(X, y)

y_pred = per_clf.predict([[2, 0.5]])
print("y_pred:",y_pred)


a = -per_clf.coef_[0][0] / per_clf.coef_[0][1]
b = -per_clf.intercept_ / per_clf.coef_[0][1]

axes = [0, 5, 0, 2]

x0, x1 = np.meshgrid(
        np.linspace(axes[0], axes[1], 500).resha
        np.linspace(axes[2], axes[3], 200).reshape(-1, 1),
    )
X_new = np.c_[x0.ravel(), x1.ravel()]
y_predict = per_clf.predict(X_new)
zz = y_predict.reshape(x0.shape)

plt.figure(figsize=(10, 4))
plt.plot(X[y==0, 0], X[y==0, 1], "bs", label="Not Iris-Setosa")
plt.plot(X[y==1, 0], X[y==1, 1], "yo", label="Iris-Setosa")

plt.plot([axes[0], axes[1]], [a * axes[0] + b, a * axes[1] + b], "k-", linewidth=3)
from matplotlib.colors import ListedColormap
custom_cmap = ListedColormap(['#9898ff', '#fafab0'])

plt.contourf(x0, x1, zz, cmap=custom_cmap, linewidth=5)
plt.xlabel("Petal length", fontsize=14)
plt.ylabel("Petal width", fontsize=14)
plt.legend(loc="lower right", fontsize=14)
plt.axis(axes)

fl.save_fig("perceptron_iris_plot")
plt.show()
```



Artificial Neural Network:Activation Functions



Artificial Neural Network:Activation Functions: Contour Plot



Artificial Neural Network:TensorFlow:HighLevel

```
WORK_HOME = os.environ['WORK_HOME']
TMP= WORK_HOME+"/resources/tmp"
fl = fileloader("")

from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets(TMP+"/data/")
X_train = mnist.train.images
X_test = mnist.test.images
y_train = mnist.train.labels.astype("int")
y_test = mnist.test.labels.astype("int")


feature_columns = tf.contrib.learn.infer_real_valued_columns_from_input(X_train)
dnn_clf = tf.contrib.learn.DNNClassifier(hidden_units=[300, 100], n_classes=10,
                                         feature_columns=feature_columns)
dnn_clf.fit(x=X_train, y=y_train, batch_size=50, steps=40000)

from sklearn.metrics import accuracy_score

y_pred = list(dnn_clf.predict(X_test))
accuracy = accuracy_score(y_test, y_pred)
print("accuracy:",accuracy)

from sklearn.metrics import log_loss

y_pred_proba = list(dnn_clf.predict_proba(X_test))
log_loss(y_test, y_pred_proba)

print(dnn_clf.evaluate(X_test, y_test))
```

Artificial Neural Network:TensorFlow:Manual-1

```python
#Construction Phase
def neuron_layer(X, n_neurons, name, activation=None):
    with tf.name_scope(name):
        n_inputs = int(X.get_shape()[1])
        stddev = 1 / np.sqrt(n_inputs)
        init = tf.truncated_normal((n_inputs, n_neurons), stddev=stddev)
        W = tf.Variable(init, name="weights")
        print("Shape of input:",X.get_shape())
        print("Shape of Weights:",W.get_shape())
        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

tf.reset_default_graph()
n_inputs = 28*28  # MNIST
n_hidden1 = 300
n_hidden2 = 100
n_outputs = 10
learning_rate = 0.01

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

with tf.name_scope("dnn"):
    hidden1 = neuron_layer(X, n_hidden1, "hidden1", activation="relu")
    hidden2 = neuron_layer(hidden1, n_hidden2, "hidden2", activation="relu")
    logits = neuron_layer(hidden2, n_outputs, "output")

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")

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

- Same as before ($1/\sqrt{\text{num inputs}}$)
- Same as before. Normal Distribution truncated by Std-dev
- shape of weights
- For layer-1
  - Input shape is

$$\begin{array}{c} \text{784} \\ \left[\;\text{num input}\;\right] \end{array} \times$$

$$\begin{array}{cc} & 300 \\ 784 & \left[\quad\right] \end{array} = \begin{array}{c} 300 \\ \left[\;\text{num inputs}\;\right] \end{array}$$

Artificial Neural Network:TensorFlow:Manual-2

```python
#Construction Phase
def neuron_layer(X, n_neurons, name, activation=None):
    with tf.name_scope(name):
        n_inputs = int(X.get_shape()[1])
        stddev = 1 / np.sqrt(n_inputs)
        init = tf.truncated_normal((n_inputs, n_neurons), stddev=stddev)
        W = tf.Variable(init, name="weights")
        print("Shape of input:",X.get_shape())
        print("Shape of Weights:",W.get_shape())
        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

tf.reset_default_graph()
n_inputs = 28*28  # MNIST
n_hidden1 = 300
n_hidden2 = 100
n_outputs = 10
learning_rate = 0.01

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

with tf.name_scope("dnn"):
    hidden1 = neuron_layer(X, n_hidden1, "hidden1", activation="relu")
    hidden2 = neuron_layer(hidden1, n_hidden2, "hidden2", activation="relu")
    logits = neuron_layer(hidden2, n_outputs, "output")

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")

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

- shape of weights
- $$\begin{array}{cc} & 300 \\ \text{num inputs} & \begin{bmatrix} \quad & \quad \end{bmatrix} \end{array} \times \begin{array}{cc} & 100 \\ 300 & \begin{bmatrix} \quad & \quad \end{bmatrix} \end{array} = \begin{array}{cc} & 100 \\ \text{num inputs} & \begin{bmatrix} \quad & \quad \end{bmatrix} \end{array}$$

Artificial Neural Network:TensorFlow:Manual-3

```python
#Construction Phase
def neuron_layer(X, n_neurons, name, activation=None):
    with tf.name_scope(name):
        n_inputs = int(X.get_shape()[1])
        stddev = 1 / np.sqrt(n_inputs)
        init = tf.truncated_normal((n_inputs, n_neurons), stddev=stddev)
        W = tf.Variable(init, name="weights")
        print("Shape of input:",X.get_shape())
        print("Shape of Weights:",W.get_shape())
        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

tf.reset_default_graph()
n_inputs = 28*28  # MNIST
n_hidden1 = 300
n_hidden2 = 100
n_outputs = 10
learning_rate = 0.01

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

with tf.name_scope("dnn"):
    hidden1 = neuron_layer(X, n_hidden1, "hidden1", activation="relu")
    hidden2 = neuron_layer(hidden1, n_hidden2, "hidden2", activation="relu")
    logits = neuron_layer(hidden2, n_outputs, "output")

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")

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

$$\begin{array}{c} \text{— 100 —} \\ \begin{bmatrix} \text{num inputs} \end{bmatrix} \end{array} \times \begin{array}{c} \text{— 10 —} \\ \begin{bmatrix} 100 \end{bmatrix} \end{array}$$

$$= \begin{array}{c} \text{— 10 —} \\ \begin{bmatrix} \text{num input} \end{bmatrix} \end{array}$$

• Into the final 10 neurons that output prob.

• Same softmax function that we used for multiclass logistic regression that penalizes low probability for target class.

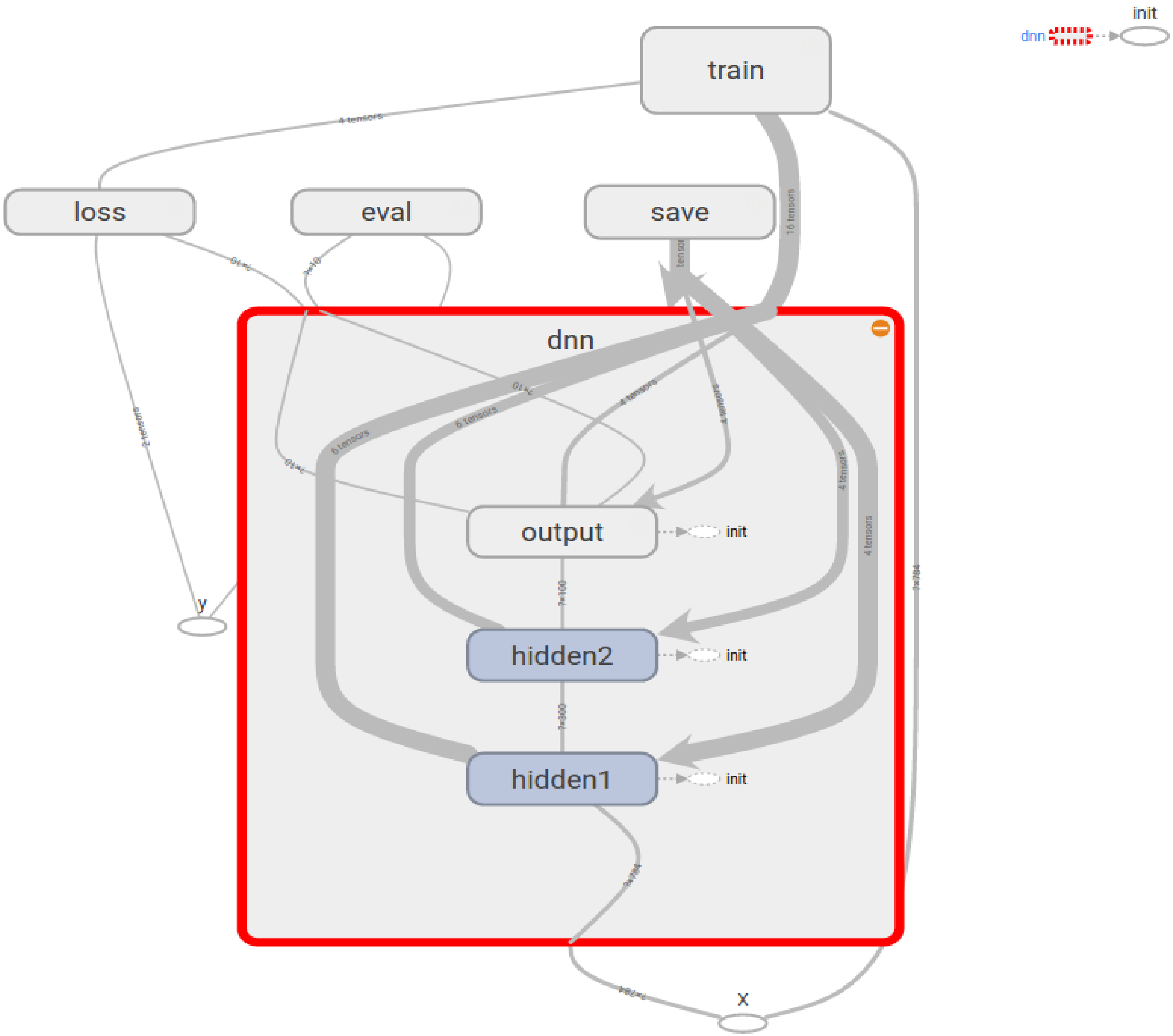
Artificial Neural Network:TensorFlow:Manual-4

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

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()

mse_summary = tf.summary.scalar('ACCURACY', accuracy)
summary_writer = tf.summary.FileWriter(logdir, tf.get_default_graph())

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


#Execution Phase
n_epochs = 20
batch_size = 50
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})
            summary_str = mse_summary.eval(feed_dict={X: X_batch, y: y_batch})
            summary_writer.add_summary(summary_str, epoch)
        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)
```

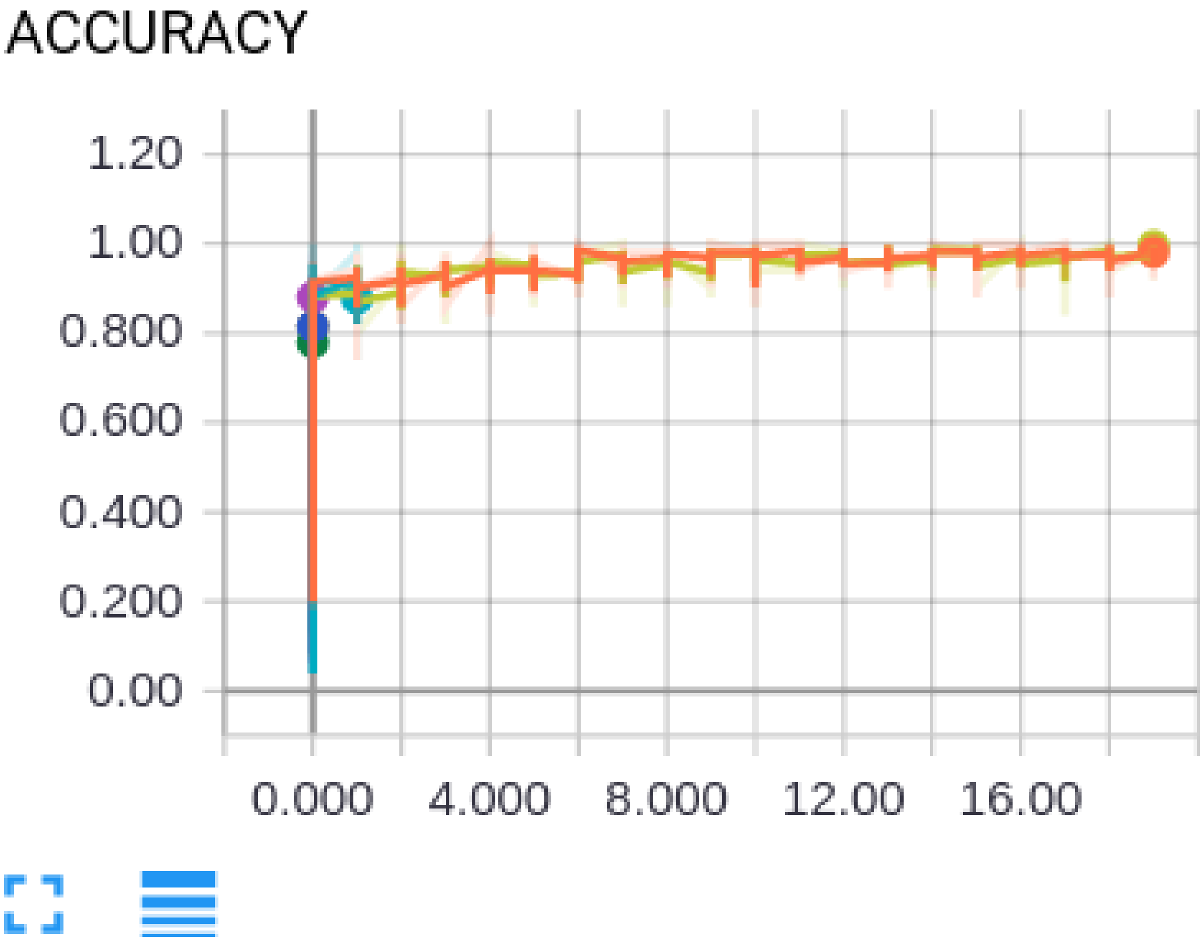
- get mini batches
- Define the feed dictionary
- Connect the input to the placeholders.
- Test the accuracy.
- Write the graph, summary, and checkpoint to a file.

- As you see, it is all matrix multiplication in the end.
- And this is what makes vector SIMD instructions and GPUs vital for NNs.
- Googles TPU is GPGPU taken to throughput extreme.

Artificial Neural Network:TensorFlow:Manual-5



Artificial Neural Network:TensorFlow:Manual-6



Artificial Neural Network:TensorFlow:Model:restore

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

with tf.name_scope("dnn"):
    hidden1 = neuron_layer(X, n_hidden1, "hidden1", activation="relu")
    hidden2 = neuron_layer(hidden1, n_hidden2, "hidden2", activation="relu")
    logits = neuron_layer(hidden2, n_outputs, "output")

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")

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

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()

mse_summary = tf.summary.scalar('ACCURACY', accuracy)
summary_writer = tf.summary.FileWriter(logdir, tf.get_default_graph())

#Execution Phase
with tf.Session() as sess:
    saver.restore(sess, root_logdir+"/my_model_final.ckpt") #"my_model_final.ckpt")
    X_new_scaled = mnist.test.images[:20]
    Z = logits.eval(feed_dict={X: X_new_scaled})
    print(np.argmax(Z, axis=1))
    print(mnist.test.labels[:20])
```

Artificial Neural Network:TensorFlow:inbuilt: fully connected

```
with tf.name_scope("dnn"):
    hidden1 = neuron_layer(X, n_hidden1, "hidden1", activation="relu")
    hidden2 = neuron_layer(hidden1, n_hidden2, "hidden2", activation="relu")
    logits = neuron_layer(hidden2, n_outputs, "outputs")
```

```
from tensorflow.contrib.layers import fully_connected

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)
```

Artificial Neural Network:Tensorflow:Finetune

- Hyperparameters
  - Grid search with cross-validation to find the right hyperparameters but since there are many hyperparameters to tune, and since training a neural network on a large dataset takes a lot of time, you will only be able to explore a tiny part of the hyperparameter space in a reasonable amount of time.
  - It is much better to use randomized search
  - Another option is to use a tool such as Oscar, which implements more complex algorithms to help you find a good set of hyperparameters quickly

Artificial Neural Network:Tensorflow:Finetune

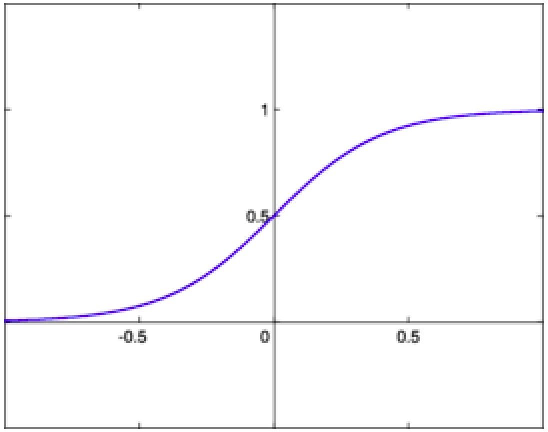
- Number of Hidden Layers
  - For a long time, these facts convinced researchers that there was no need to investigate any deeper neural networks.
  - But they overlooked the fact that deep networks have a much higher parameter efficiency than shallow ones: **they can model complex functions using exponentially fewer neurons than shallow nets, making them much faster to train**.
  - For many problems you can start with just one or two hidden layers and it will work just fine (e.g., you can easily reach above 97% accuracy on the MNIST dataset using just one hidden layer with a few hundred neurons, and above 98% accuracy using two hidden layers with the same total amount of neurons, in roughly the same amount of training time).
  - For more complex problems, you can gradually ramp up the number of hidden layers, until you start overfitting the training set. Very complex tasks, **such as large image classification or speech recognition, typically require networks with dozens of layers like (CNNs)**

Artificial Neural Network:Tensorflow:Finetune

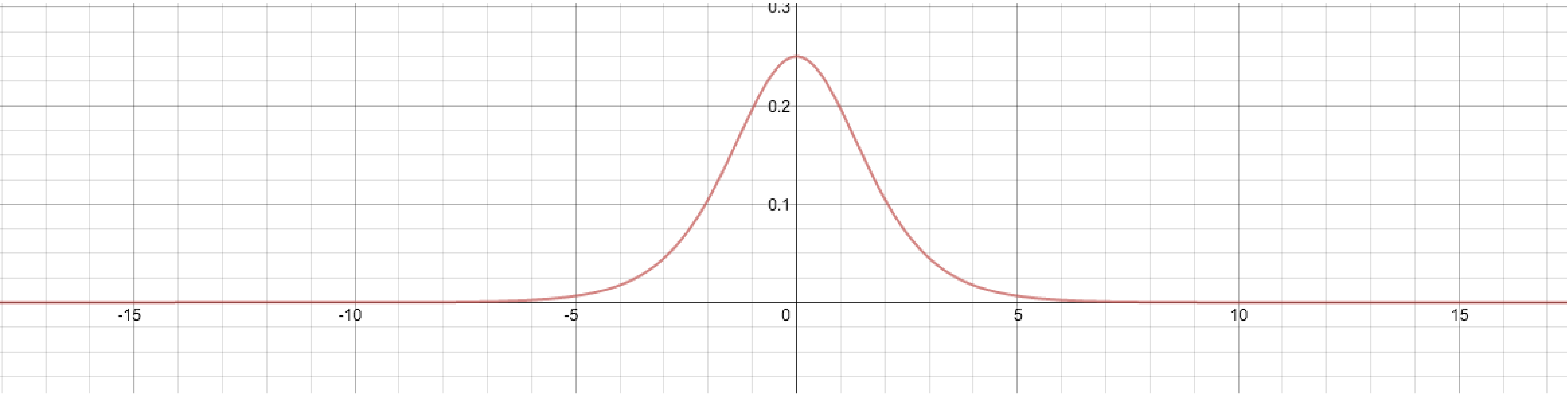
- Number of neurons/layer
  - The hidden layers, a common practice is to size them to form a funnel, with fewer and fewer neurons at each layer—the rationale being that many low-level features can coalesce into far fewer high-level features. For example, a typical neural network for MNIST may have two hidden layers, the first with 300 neurons and the second with 100.
  - However, this practice is not as common now, and you may simply use the same size for all hidden layers—for example, all hidden layers with 150 neurons: that's just one hyperparameter to tune instead of one per layer.
  - Just like for the number of layers, you can try increasing the number of neurons gradually until the network starts overfitting.
  - In general you will get more bang for the buck by **increasing the number of layers than the number of neurons per layer**. Unfortunately, as you can see, finding the perfect amount of neurons is still somewhat of a **black art**.
  - A simpler approach is to pick a model with more layers and neurons than you actually need, then use **early stopping** to prevent it from overfitting (and other **regularization techniques, especially dropout**).
  - This has been dubbed the "stretch pants" approach:12 instead of wasting time looking for pants that perfectly match your size, just use large stretch pants that will shrink down to the right size.

Artificial Neural Network:Problems with Activation Function

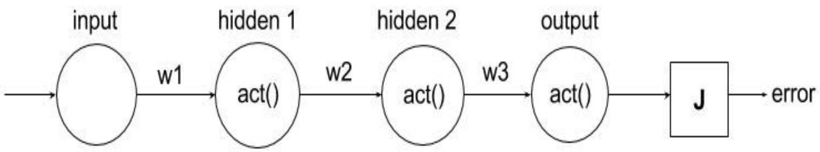
$$Sigmoid = S(\alpha) = \frac{1}{1+e^{-\alpha}}$$



$$\frac{1}{1+e^{-\alpha}}\left[1-\frac{1}{1+e^{-\alpha}}\right]$$



Artificial Neural Network:Problems with Activation Function :Vanishing/Exploding gradient



$$\frac{\partial error}{\partial w1} = \frac{\partial error}{\partial output} * \frac{\partial output}{\partial hidden2} * \frac{\partial hidden2}{\partial hidden1} * \frac{\partial hidden1}{\partial w1}$$

$$\frac{\partial output}{\partial hidden2} * \frac{\partial hidden2}{\partial hidden1}$$

$$\boxed{z_1 = hidden2 * w3}$$

$$\frac{\partial output}{\partial hidden2} = \frac{\partial Sigmoid(z_1)}{\partial z_1} w3$$

$$z_2 = hidden1 * w2$$

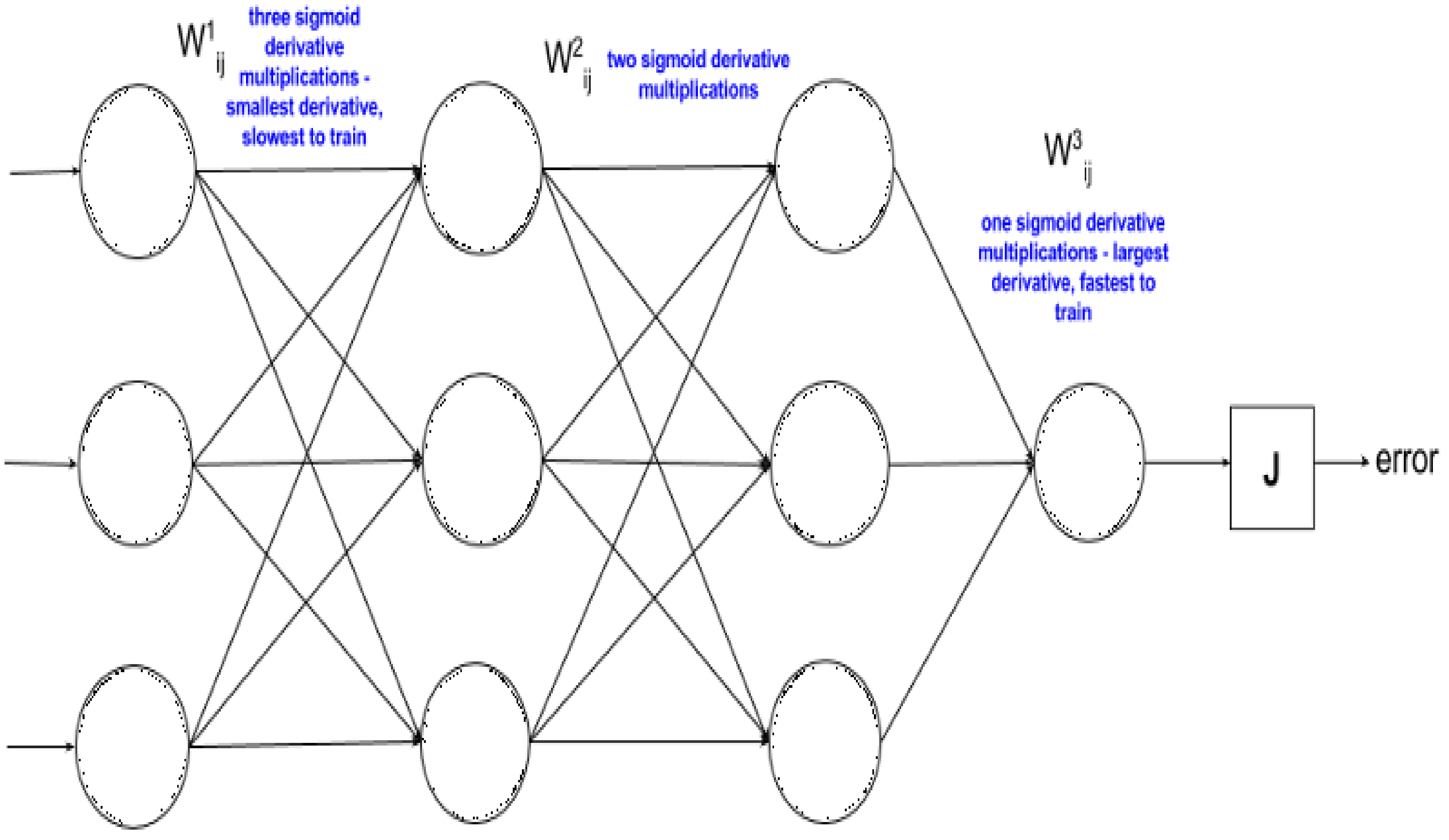
$$\frac{\partial hidden2}{\partial hidden1} = \frac{\partial Sigmoid(z_2)}{\partial z_2} w2$$

$$\frac{\partial output}{\partial hidden2}\frac{\partial hidden2}{\partial hidden1} = \frac{\partial Sigmoid(z_1)}{\partial z_1} w3 * \frac{\partial Sigmoid(z_2)}{\partial z_2} w2$$

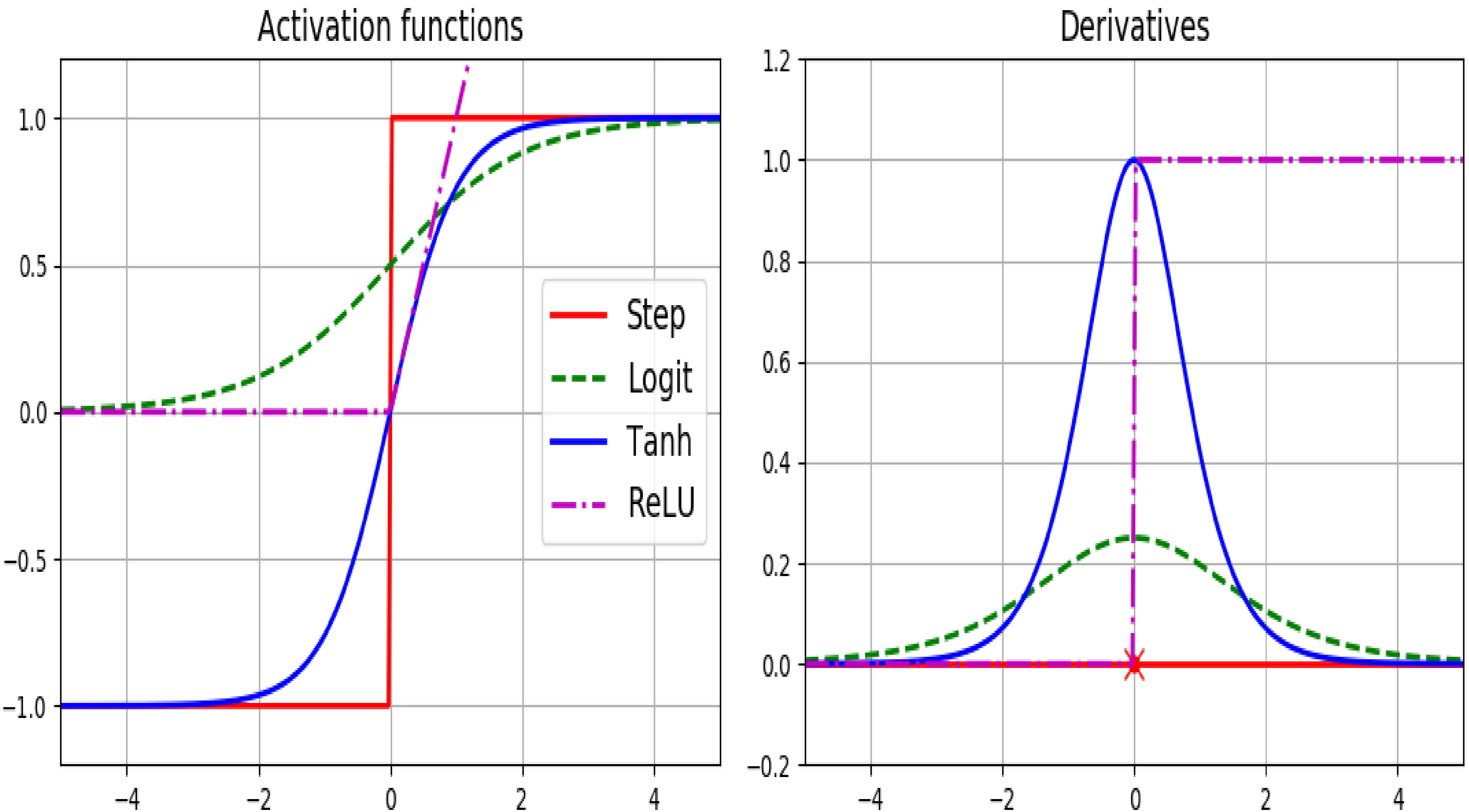
$$\frac{\partial output}{\partial hidden2}\frac{\partial hidden2}{\partial hidden1} = \overset{<1/4}{\frac{\partial Sigmoid(z_1)}{\partial z_1}} \overset{<1}{w3} * \overset{<1/4}{\frac{\partial Sigmoid(z_2)}{\partial z_2}} \overset{<1}{w2}$$

- We have derived this earlier, refer to deep neural network slide part-2.
- W X previos layer input
- We are muliplying those small numbers to calculate the gradient
- This will become very small very fast, specially for deeper networks.
- For a deeper network the gradient almost vanishes the further back we propagate on the backward pass.

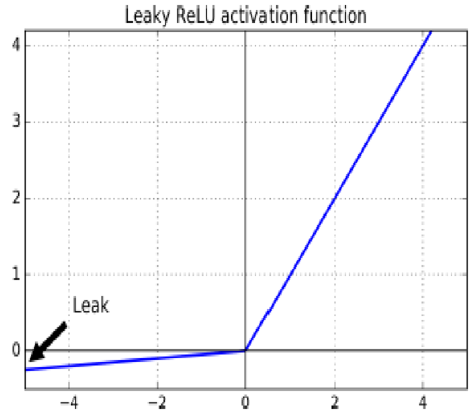
Artificial Neural Network:Problems with Activation Function :Vanishing/Exploding gradient



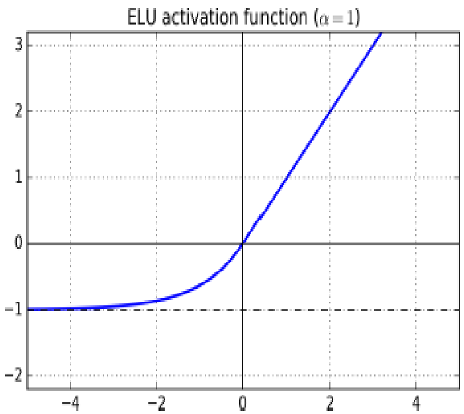
Artificial Neural Network:Problems with Activation Function :Vanishing/Exploding gradient



Artificial Neural Network:Problems with Activation Function :Vanishing/Exploding gradient



leaky relu, avoid the problem of negative input of relu.
Both these are computationally light weight compared to sigmoid.



- ELU or exponential linear unit out performs all variants of RELU.

$$ELU_\alpha(z) = \begin{cases} \alpha(\exp(z) - 1) & \text{if } z < 0 \\ z & \text{if } z > 0 \end{cases}$$

Artificial Neural Network:Problems with Activation Function :Vanishing/Exploding gradient:TensorFlow:LeakyRelu

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

Artificial Neural Network:Problems with Activation Function :Vanishing/Exploding gradient:TensorFlow:elu

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

Artificial Neural Network:Problems with Activation Function :Vanishing/Exploding gradient:Batch Normalization

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

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

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

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

- 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.

- This is done before the activation function (LeakyRelu or ELU) is called.
- This is more for overall accuracy but also helps with vanishing gradient.

Artificial Neural Network:Problems with Activation Function :Vanishing/Exploding gradient:tensorflow:Batch Normalization

```python
with tf.name_scope("dnn"):
    he_init = tf.contrib.layers.variance_scaling_initializer()

    my_batch_norm_layer = partial(
            tf.layers.batch_normalization,
            training=is_training,
            momentum=0.9)

    my_dense_layer = partial(
            tf.layers.dense,
            kernel_initializer=he_init)

    hidden1 = my_dense_layer(X, n_hidden1, name="hidden1")
    bn1 = tf.nn.elu(my_batch_norm_layer(hidden1))
    hidden2 = my_dense_layer(bn1, n_hidden2, name="hidden2")
    bn2 = tf.nn.elu(my_batch_norm_layer(hidden2))
    logits_before_bn = my_dense_layer(bn2, n_outputs, activation=None, name="outputs")
    logits = my_batch_norm_layer(logits_before_bn)
    extra_update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
```

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)	$r = \sqrt{2}\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{2}\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$

- a simplified code used in an earlier example
- consult partial functions in python

$X \cdot W \rightarrow BN \rightarrow$ Activation

Artificial Neural Network:Problems with Activation Function :Vanishing/Exploding gradient:tensorflow:Batch Normalization

```
with tf.name_scope("dnn"):
    he_init = tf.contrib.layers.variance_scaling_initializer()

    my_batch_norm_layer = partial(
            tf.layers.batch_normalization,
            training=is_training,
            momentum=0.9)

    my_dense_layer = partial(
            tf.layers.dense,
            kernel_initializer=he_init,
            kernel_regularizer=tf.contrib.layers.l1_regularizer(0.01))

    hidden1 = my_dense_layer(X, n_hidden1, name="hidden1")
    bn1 = tf.nn.elu(my_batch_norm_layer(hidden1))
    hidden2 = my_dense_layer(bn1, n_hidden2, name="hidden2")
    bn2 = tf.nn.elu(my_batch_norm_layer(hidden2))
    logits_before_bn = my_dense_layer(bn2, n_outputs, activation=None, name="outputs")
    logits = my_batch_norm_layer(logits_before_bn)
    extra_update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
```

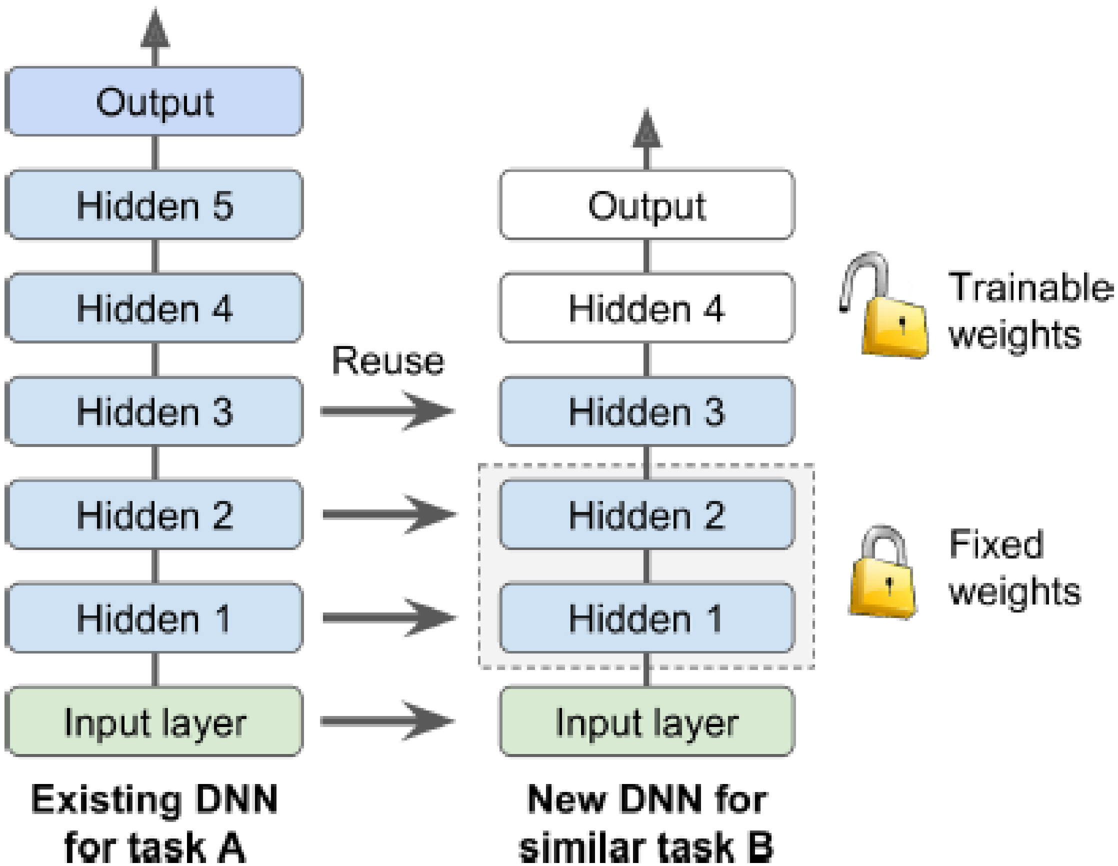
- Same as before except for l_1 regularization term.
- Also BN is unlikely to effect a shallow network but will effect deep networks positively

Artificial Neural Network:Problems with Activation Function :Vanishing/Exploding gradient:tensorflow:GradientClipping

```
threshold = 1.0
optimizer = tf.train.GradientDescentOptimizer(learning_rate)
grads_and_vars = optimizer.compute_gradients(loss)
capped_gvs = [(tf.clip_by_value(grad, -threshold, threshold), var)
              for grad, var in grads_and_vars]
training_op = optimizer.apply_gradients(capped_gvs)
```

- Only for completness as BN works better in general.

Artificial Neural Network:Pre Trained Layers



Artificial Neural Network:Freezing Lower layers

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

- Likely that lower layer of first DNN have learn to detect low level features that will be useful across both image classification tasks.

Artificial Neural Network:Caching Frozen Layers

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

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})
```

- Since the frozen layers won't change, it is possible to cache it provided the memory is enough.
- In this case Tensorflow does not evaluate the output from hidden-2 or any node that it depends on.

Artificial Neural Network:Tweaking, Dropping, or Replacing the Upper Layers

- Try freezing all the copied layers first, then train your model and see how it performs.
- Then try unfreezing one or two of the top hidden layers to let backpropagation tweak them and see if performance improves.
- The more training data you have, the more layers you can unfreeze.
- If you still cannot get good performance, and you have little training data, try dropping the top hidden layer(s) and freeze all remaining hidden layers again

Artificial Neural Network:Finetuning

- Model Zoos
  - TensorFlow has its own model zoo available at https://github.com/tensorflow/models.
    - In particular, it contains most of the state-of-the-art image classification nets such as VGG, Inception, and ResNet
  - Another popular model zoo is Caffe’s Model Zoo. It also contains many computer vision models (e.g., LeNet, AlexNet, ZFNet, GoogLeNet, VGGNet, inception) trained on various datasets (e.g., ImageNet, Places Database, CIFAR10, etc.).
    - Saumitro Dasgupta wrote a converter, which is available at https://github.com/ethereon/caffetensorflow.

Artificial Neural Network:Finetuning

- Pretraining on Auxiliary Task
  - if you want to build a system to recognize faces, you may only have a few pictures of each individual—clearly not enough to train a good classifier.
  - Gathering hundreds of pictures of each person would not be practical. However, you could gather a lot of pictures of random people on the internet and train a first neural network to detect whether or not two different pictures feature the same person.
  - Such a network would learn good feature detectors for faces, so reusing its lower layers would allow you to train a good face classifier using little training data.
  - It is often rather cheap to gather unlabeled training examples, but quite expensive to label them.

Artificial Neural Network:Finetuning

- Optimizers
  - Momemtum
  - Nesterov Accelerated Gradient
  - AdaGrad
  - RMSProp
  - Adam

Artificial Neural Network:Finetuning:Overfitting

- Regularization
  - L1 and L2
- Early Stopping
- Dropouts
- Data Augmentation-Rotation