1. **Is it OK to initialize all the weights to the same value as long as that value is selected randomly using He initialization?**

**ANS:-**

No. All weights should be initialized to different random values and should not have the same initial value. If weights are symmetrical, meaning they have the same value, it makes it almost impossible for backpropagation to converge to a good solution.

Think of it this way: if all the weights are the same, it's like having just one neuron per layer, but **much** slower.

The technique we use to break this symmetry is to sample weights randomly.

1. **Is it OK to initialize the bias terms to 0?**

**ANS:-** Yes, this is fine. It does not end up making much of a difference. It's been my practice to initialize the bias term to 1.

1. **Name three advantages of the SELU activation function over ReLU.**

**ANS:-**

* It can take on negative values, so the average output of the neurons in any given layer is typically closer to 0 then when using the ReLU function. This helps alleviate the vanishing gradients problem. The vanishing gradients problem is the idea that gradients often get smaller and smaller as the algorithm progresses down to the lower layers. As a result, the gradient descent update leaves the lower layer connection weights virtually unshanged, and training never converges to a good solution.
* It always has a non-zero derivative, which avoids the dying units issue that can affect ReLU units. "Dying ReLU's" is the problem where units stop outputting anything other than 0. In some cases, you may find that half of your networks neurons are dead, especially if you use a large learning rate.
* It is smooth everywhere, which helps gradient descent converge faster. ReLU's slope abruptly jumps from 0 to 1 at z = 0. Such an abrupt change can slow down gradient descent because it will bounce around z = 0.

1. **In which cases would you want to use each of the following activation functions: SELU, leaky ReLU (and its variants), ReLU, tanh, logistic, and softmax?**

**ANS:-**

The ELU function is a good default.

If you need the network to be as fast as possible, you can use one of the leaky ReLU variants.

The basic ReLU function is also preferred due to its simplicity, despite the fact they are generally outperformed by ELU and leaky ReLU. If you have a situation where it's preferable to have neurons output exactly 0, then ReLU is a good choice.

tanh can be useful in the output layer if you need your outputs to be between -1 and 1. It's not used too much in hidden layers.

The logistic function is useful in the output layer when you need to estimate a probability like in the binary, multi-class, or multi-class multi-label classification problems. Like tanh, it is not used in the hidden layers.

The softmax function is useful in the output layer when you need to output probabilities for mutually exclusive classes. Again, it's not used in hidden layers.

1. **What may happen if you set the momentum hyperparameter too close to 1 (e.g., 0.99999) when using an SGD optimizer?**

**ANS:-**

The purpose of the momentum hyperparameter B is to simulate friction and prevent the momentum from growing too large. We set it to 0 for high friction, and 1 for no friction.

As the hyperparameter gets closer to 1, there will be less friction and the momentum optimization will "roll faster down the hill." This means that the optimizer will overshoot, then come back, overshoot again, and oscillate like this many times before stabilizing at the minimum. This is one of the reasons why it is good to have a bit of friction in the system: it gets rid of these oscillations and thus speeds up convergence.

1. **Name three ways you can produce a sparse model.**

**ANS:-**

A sparse model is that where most weights are equal to 0. There's a couple ways of achieving that effect.

You can train the model normally then zero out tiny weights.

For more sparsity, you can apply l1 regularization during training, which pushes the optimizer towards sparsity.

Finally, you can combine l1 regulatization with **dual averaging** using TensorFlow's FTRLOptimizer class.

1. **Does dropout slow down training? Does it slow down inference (i.e., making predictions on new instances)? What about MC Dropout?**

**ANS:-**

Dropout is a popular regularization technique for deep neural networks.

The algorithm is: at each training step, every neuron (including the input neurons but excluding the output neurons) has a probability p of being temporariliy "dropped out", meaning it will be entierly ignored during this training step. However, it may be active during the next step.

The hyperparameter p is called the dropout rate, and it is typically set to 0.5.

After training the neurons don't get dropped anymore. That's the gist of the algorithm.

Dropout does tend to significantly slow down convergence, but it usually results in a much better model when tuned properly. Remember, because dropout is only tuned during training, it has no impact on inference.

**Deep Learning**

* Build a DNN with five hidden layers of 100 neurons each, using He initialization for the weights and the ELU activation function.
* Using Adam optimization and early stopping, try training MNIST but only on digits 0 to 4. (We will use transfer learning for digits 5 to 9 in the next exercise.) You will need a softmax output layer with 5 neurons, and as always make sure to save checkpoints at regular intervals and save the final model so you can reuse it later.
* Tune the hyperparameters using cross-validation and see what precision you can achieve.
* Now try adding batch normalization and compare the learning curves: is it converging faster than before? Does it produce a better model?
* Is the model overfitting the training set? Try adding dropout to every layer and try again. Does it help?

**Transfer Learning**

* Create a new DNN that reuses all the pretrained hidden layers of the previous model, freezes them, and replaces the softmax output layer with a fresh new one.
* Train this new DNN on digits 5 to 9, using only 100 images per digit, and time how long it takes. Despite this small number of examples, can you achieve high precision?
* Try caching the frozen layers, and train the model again: how much faster is it now?
* Try again reusing just four hidden layers instead of five. Can you achieve a higher precision?
* Now unfreeze the top two hidden layers and continue training: can you get the model to perform even better?

Monte Carlo Dropout was introduced in a**2016 research paper** by ***Yarin Gal and Zoubin Ghahramani***, is a technique that combines two powerful concepts in machine learning: **Monte Carlo methods** and **dropout regularization.** This innovation can be thought of as an upgrade to traditional dropout, offering the potential for significantly more accurate predictions. It is done is at time of **testing** . In this article, we’ll delve into the concepts and workings of Monte Carlo Dropout.

## ****Understanding Dropout****

Dropout is primarily used as a ***regularization technique***, a method employed to fine-tune machine learning models. It aims to optimize the adjusted loss function while avoiding the issues of overfitting or underfitting. When implemented, traditional dropout typically results in a modest increase in model accuracy, usually in the range of 1% to 2%. This improvement is credited to its effectiveness in reducing overfitting, which, in turn, minimizes errors in the model’s predictions.

The dropout rate typically falls within the range of 0 (signifying no dropout) to 0.5 (meaning around 50% of all neurons will be deactivated). The specific value is determined by factors such as the type of network, the size of its layers, and the extent to which the network tends to overfit the training data. At every iteration different set of neurons are dropped, corresponding with its ingoing and outgoing directions.

## Monte Carlo Dropout

The Monte Carlo Dropout technique, as introduced by Gal and Ghahramani in 2016, involves estimation of uncertainty in predictions made by models. By applying dropout at test time and running multiple forward passes with different dropout masks, the model produces a **distribution of predictions** rather than a single point estimate. This distribution provides insights into the model’s uncertainty about its predictions, effectively regularizing the network.

**Step 1: Estimation of Predictive Distribution**

It leverages the principles of Bayesian neural networks to estimate uncertainty and improve the predictive capabilities of neural network models. One of the primary goals of Monte Carlo Dropout is to obtain a predictive distribution, which allows for the uncovering of uncertainty in the model’s predictions. The **predictive distribution** is a probability distribution that represents the uncertainty about the value of a future observation or outcome given the data, observed.

To achieve this, Monte Carlo Dropout employs the concept of learning a distribution over functions, or equivalently, a distribution over the parameters, known as the parametric posterior distribution. This distribution captures the uncertainty in the model’s parameters and enables the exploration of different network configurations.

Where,  represents dropout mask, sampled from  approximate parametric posterior.

**Step 2: Exploration of Diverse Network Configurations**

Each dropout configuration corresponds to a different sample from the approximate parametric posterior distribution. This process allows for the exploration of diverse network configurations and facilitates the learning of a predictive distribution.

**Step 3: Monte Carlo Integration for Uncovering Predictive Distribution**

By sampling from the approximate posterior distribution, Monte Carlo Dropout enables Monte Carlo integration of the model’s likelihood, leading to the uncovering of the predictive distribution. This predictive distribution provides insights into the uncertainty associated with the model’s predictions, allowing for more informed decision-making.

However, the likelihood may be assumed to be Gaussian distributed for simplicity.

The Gaussian function, denoted by  is characterized by the mean

 and variance

parameters, which are obtained from simulations using the Monte Carlo dropout Bayesian neural network (BNN).

**Step 4:** **Estimation of Uncertainty and Reliable Predictions**

This approach allows for the estimation of uncertainty and the generation of more reliable predictions, making the model more robust and dependable.

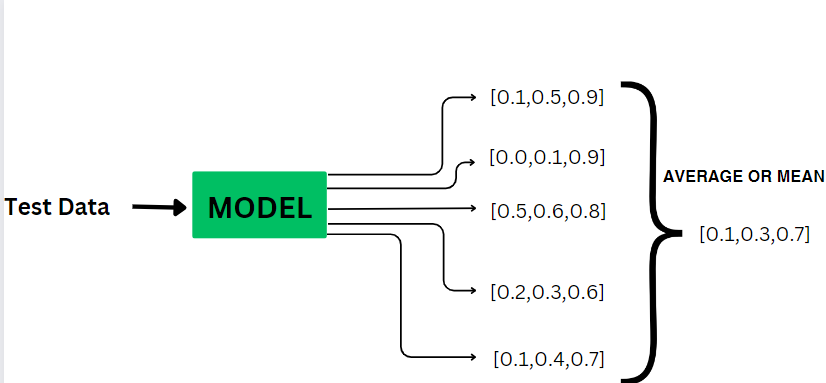
## Applying Dropout During Testing-Monte Carlo

The process of Monte Carlo Dropout during testing ***involves two key steps***:

1.**For predicating test data we’ll keep drop out data activated :** For predicting test data, we keep dropout in effect. This means that different sets of neurons are deactivated at each iteration.

Model(x\_test,Training=True)

**2.Perform any simulation to find ‘T’ output for test data :**For every test data that we are passing through the model we’ll try to get some T results i.e. T specifies my simulation . Each dropout model provides different scores, and then we compute the average of these scores. The final prediction is the average of predictions from .



**Mathematically,**

In the equation:

T represents the number of forward passes or samples we take with different dropout masks. Each forward pass generates a prediction, denoted as **.**

Dropout masks refer to the random patterns of dropout applied to the neurons in a neural network during each forward pass. By using**different dropout masks** for each forward pass, the network is forced to learn more robust representations that are not overly dependent on specific neurons. This helps improve the generalization ability of the network and reduces the risk of overfitting. The sum of all the predictions, , represents the cumulative output of the network across all the forward passes.

The term**() is a scaling factor**that ensures the final prediction is an average of the individual predictions. It divides the sum of the predictions by the total number of forward passes, T.

The final prediction, **y\_final**, represents the average of the T predictions and can be interpreted as the expected value or mean prediction of the network. It provides a more reliablе estimate compared to a single prediction obtained without dropout

1. **Practice training a deep neural network on the CIFAR10 image dataset:**
   1. **Build a DNN with 20 hidden layers of 100 neurons each (that’s too many, but it’s the point of this exercise). Use He initialization and the ELU activation function.**

**ANS:-**

keras.backend.clear\_session()

tf.random.set\_seed(42)

np.random.seed(42)

# Model

model = keras.models.Sequential()

# Input layer

model.add(keras.layers.Flatten(input\_shape=[32, 32, 3]))

# Hidden Layers

for \_ in range(20):

    model.add(keras.layers.Dense(

        100,

        activation='elu',

        kernel\_initializer='he\_normal'

    ))

* 1. **Using Nadam optimization and early stopping, train the network on the CIFAR10 dataset. You can load it with keras.datasets.cifar10.load\_​data(). The dataset is composed of 60,000 32 × 32–pixel color images (50,000 for training, 10,000 for testing) with 10 classes, so you’ll need a softmax output layer with 10 neurons. Remember to search for the right learning rate each time you change the model’s architecture or hyperparameters.**

**ANS:-**

# Output Layer

model.add(keras.layers.Dense(10, activation='softmax'))

optimizer = keras.optimizers.Nadam(learning\_rate=5e-5)

model.compile(

    loss='sparse\_categorical\_crossentropy',

    optimizer=optimizer,

    metrics=['accuracy']

)

(X\_train\_full, y\_train\_full), (X\_test, y\_test) = keras.datasets.cifar10.load\_data()

X\_train = X\_train\_full[5000:]

y\_train = y\_train\_full[5000:]

X\_valid = X\_train\_full[:5000]

y\_valid = y\_train\_full[:5000]

run\_index = 1 # increment every time you train the model

run\_logdir = os.path.join(os.curdir, "my\_cifar10\_logs", "run\_{:03d}".format(run\_index))

def get\_run\_logdir():

  import time

  run\_id = time.strftime("run\_%Y\_%m\_%d-%H\_%M\_%S")

  return os.path.join(run\_logdir, run\_id)

run\_logdir = get\_run\_logdir()  # e.g., './my\_logs/run\_2019\_06\_07-15\_15\_22'

# [...] # Build and compile your model

tensorboard\_cb = keras.callbacks.TensorBoard(run\_logdir)

history = model.fit(X\_train, y\_train, epochs=30,

                    validation\_data=(X\_valid, y\_valid),

                    callbacks=[tensorboard\_cb])

* 1. **Now try adding Batch Normalization and compare the learning curves: Is it converging faster than before? Does it produce a better model? How does it affect training speed?**

**ANS:-**

*# extra code - clear the name counters and set the random seed*

tf**.**keras**.**backend**.**clear\_session()

tf**.**random**.**set\_seed(42)

model **=** tf**.**keras**.**Sequential([

tf**.**keras**.**layers**.**Flatten(input\_shape**=**[28, 28]),

tf**.**keras**.**layers**.**BatchNormalization(),

tf**.**keras**.**layers**.**Dense(300, activation**=**"relu",

kernel\_initializer**=**"he\_normal"),

tf**.**keras**.**layers**.**BatchNormalization(),

tf**.**keras**.**layers**.**Dense(100, activation**=**"relu",

kernel\_initializer**=**"he\_normal"),

tf**.**keras**.**layers**.**BatchNormalization(),

tf**.**keras**.**layers**.**Dense(10, activation**=**"softmax")

])

model**.**summary()

* 1. **Try replacing Batch Normalization with SELU, and make the necessary adjustements to ensure the network self-normalizes (i.e., standardize the input features, use LeCun normal initialization, make sure the DNN contains only a sequence of dense layers, etc.).**

**ANS:-**

* 1. **Try regularizing the model with alpha dropout. Then, without retraining your model, see if you can achieve better accuracy using MC Dropout.**

**ANS:-**

tf**.**random**.**set\_seed(42)

np**.**random**.**seed(42)

model **=** keras**.**models**.**Sequential([

keras**.**layers**.**Flatten(input\_shape**=**[28, 28]),

keras**.**layers**.**AlphaDropout(rate**=**0.2),

keras**.**layers**.**Dense(300, activation**=**"selu", kernel\_initializer**=**"lecun\_normal"),

keras**.**layers**.**AlphaDropout(rate**=**0.2),

keras**.**layers**.**Dense(100, activation**=**"selu", kernel\_initializer**=**"lecun\_normal"),

keras**.**layers**.**AlphaDropout(rate**=**0.2),

keras**.**layers**.**Dense(10, activation**=**"softmax")

])

optimizer **=** keras**.**optimizers**.**SGD(learning\_rate**=**0.01, momentum**=**0.9, nesterov**=True**)

model**.**compile(loss**=**"sparse\_categorical\_crossentropy", optimizer**=**optimizer, metrics**=**["accuracy"])

n\_epochs **=** 20

history **=** model**.**fit(X\_train\_scaled, y\_train, epochs**=**n\_epochs,

validation\_data**=**(X\_valid\_scaled, y\_valid))