# DEEP LEARNING – WORKSHEET 3

## Q1 to Q8 are MCQs with only one correct answer. Choose the correct option.

1. Which of the following is true about model capacity (where model capacity means the ability of neural network to approximate complex functions)?
   1. As dropout ratio increases, model capacity increases
   2. As number of hidden layers increase, model capacity increases
   3. As learning rate increases, model capacity increases
   4. None of the above
2. Batch Normalization is helpful because?
   1. It is a very efficient backpropagation technique
   2. It returns back the normalized mean and standard deviation of weights
   3. It normalizes (changes) all the input before sending it to the next layer
   4. None of the above
3. What if we use a learning rate that’s too large?
   1. Network will not converge B) Network will converge

C) either A or B D) None of the above

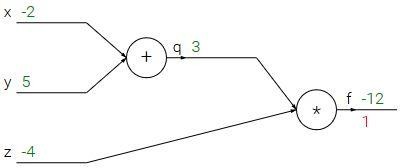
1. What are the factors to select the depth of neural network?
2. Type of neural network (e.g. MLP, CNN etc.)
3. Input data
4. Computation power, i.e. Hardware capabilities and software capabilities
5. Learning Rate
6. The output function to map

A) 1, 2, 4, 5 B) 2, 3, 4, 5

C) 1, 3, 4, 5 D) All of these

1. Suppose you have inputs as x, y, and z with values -2, 5, and -4 respectively. You have a neuron ‘q’ and neuron ‘f’ with functions:

q = x + y f = q \* z

Graphical representation of the functions is as follows:

What is the gradient of F with respect to x, y, and z? (use chain rule of derivatives to find the solution) A) (3, -4, -4) B) (-3, 4, 4)

C) (-4, -4, 3) D) (4, 4, 3)

1. Which of the following statement is the best description of early stopping?
   1. Train the network until a local minimum in the error function is reached
   2. Simulate the network on a test dataset after every epoch of training. Stop training when the generalization error starts to increase
   3. Add a momentum term to the weight update in the Generalized Delta Rule, so that training converges more quickly
   4. None of the above
2. Which gradient descent technique is more advantageous when the data is too big to handle in RAM simultaneously?
   1. Mini Batch Gradient Descent B) Stochastic Gradient Descent

C) Full Batch Gradient Descent D) either A or B

1. Consider the scenario. The problem you are trying to solve has a small amount of data. Fortunately, you have a pre-trained neural network that was trained on a similar problem. Which of the following methodologies would you choose to make use of this pre-trained network?
   1. Freeze all the layers except the last, re-train the last layer
   2. Assess on every layer how the model performs and only select a few of them
   3. Fine tune the last couple of layers only
   4. Re-train the model for the new dataset

## Q9 and Q10 are MCQs with one or more correct answers. Choose all the correct options.

1. Which of the following neural network training challenge can be solved using batch normalization?
   1. Overfitting B) Training is too slow
2. Restrict activations to become too high or low
3. None of these
4. For a binary classification problem, which of the following activations may be used in output layer?
   1. ReLU B) sigmoid

C) softmax D) Leaky ReLU

## Q11 to Q15 are subjective answer type question. Answer them briefly.

1. What will happen if we do not use activation function in artificial neural networks?

**Vanishing Gradient problem:**Neural Networks are trained using the process gradient descent. The gradient descent consists of the backward propagation step which is basically chain rule to get the change in weights in order to reduce the loss after every epoch. Consider a two-layer network and the first layer is represented as f₁(x) and the second layer is represented as f₂(x). The overall network is o(x) = f₂(f₁(x)). If we calculate weights during the backward pass, we get o`(x) = f₂(x)\*f₁`(x). Here f₁(x) is itself a compound function consisting of *Act*(W₁\*x₁ + b₁) where *Act* is the activation function after layer 1. Applying chain rule again, we clearly see that f₁`(x) = *Act*(W₁\*x₁ + b₁)\*x₁ which means it also depends directly on the activation value. Now imagine such a chain rule going through multiple layers while backpropagation. If the value of *Act*() is between 0 and 1, then several such values will get multiplied to calculate the gradient of the initial layers. This reduces the value of the gradient for the initial layers and those layers are not able to learn properly. In other words, their gradients tend to vanish because of the depth of the network and the activation shifting the value to zero. This is called the **vanishing gradient problem**. So we want our activation function to not shift the gradient towards zero.

**Zero-Centered:**Output of the activation function should be symmetrical at zero so that the gradients do not shift to a particular direction.

**Computational Expense**: Activation functions are applied after every layer and need to be calculated millions of times in deep networks. Hence, they should be computationally inexpensive to calculate.

**Differentiable:**As mentioned, neural networks are trained using the gradient descent process, hence the layers in the model need to differentiable or at least differentiable in parts. **This is a necessary requirement for a function to work as activation function layer.**

1. How does forward propagation and backpropagation work in deep learning?

## Forward Propagation

Forward propagation (or forward pass) refers to the calculation and storage of intermediate variables (including outputs) for a neural network in order from the input layer to the output layer. We now work step-by-step through the mechanics of a neural network with one hidden layer. This may seem tedious but in the eternal words of funk virtuoso James Brown, you must “pay the cost to be the boss”.

For the sake of simplicity, let us assume that the input example is x∈Rdx∈Rd and that our hidden layer does not include a bias term. Here the intermediate variable is:

z=W(1)x,z=W(1)x,

where W(1)∈Rh×dW(1)∈Rh×d is the weight parameter of the hidden layer. After running the intermediate variable z∈Rhz∈Rh through the activation function ϕϕ we obtain our hidden activation vector of length hh,

h=ϕ(z).h=ϕ(z).

The hidden variable hh is also an intermediate variable. Assuming that the parameters of the output layer only possess a weight of W(2)∈Rq×hW(2)∈Rq×h, we can obtain an output layer variable with a vector of length qq:

o=W(2)h.o=W(2)h.

Assuming that the loss function is ll and the example label is yy, we can then calculate the loss term for a single data example,

L=l(o,y).L=l(o,y).

According to the definition of L2L2 regularization, given the hyperparameter λλ, the regularization term is

s=λ2(∥W(1)∥2F+∥W(2)∥2F),s=λ2(‖W(1)‖F2+‖W(2)‖F2),

where the Frobenius norm of the matrix is simply the L2L2 norm applied after flattening the matrix into a vector. Finally, the model’s regularized loss on a given data example is:

J=L+s.J=L+s.

We refer to JJ as the objective function in the following discussion.

## 4.7.3. Backpropagation

Backpropagation refers to the method of calculating the gradient of neural network parameters. In short, the method traverses the network in reverse order, from the output to the input layer, according to the chain rule from calculus. The algorithm stores any intermediate variables (partial derivatives) required while calculating the gradient with respect to some parameters. Assume that we have functions Y=f(X)Y=f(X) and Z=g(Y)Z=g(Y), in which the input and the output X,Y,ZX,Y,Z are tensors of arbitrary shapes. By using the chain rule, we can compute the derivative of ZZ with respect to XX via

(4.7.7)

∂Z∂X=prod(∂Z∂Y,∂Y∂X).∂Z∂X=prod(∂Z∂Y,∂Y∂X).

Here we use the prodprod operator to multiply its arguments after the necessary operations, such as transposition and swapping input positions, have been carried out. For vectors, this is straightforward: it is simply matrix-matrix multiplication. For higher dimensional tensors, we use the appropriate counterpart. The operator prodprod hides all the notation overhead.

Recall that the parameters of the simple network with one hidden layer, whose computational graph is in [Fig. 4.7.1](https://d2l.ai/chapter_multilayer-perceptrons/backprop.html#fig-forward), are W(1)W(1) and W(2)W(2). The objective of backpropagation is to calculate the gradients ∂J/∂W(1)∂J/∂W(1) and ∂J/∂W(2)∂J/∂W(2). To accomplish this, we apply the chain rule and calculate, in turn, the gradient of each intermediate variable and parameter. The order of calculations are reversed relative to those performed in forward propagation, since we need to start with the outcome of the computational graph and work our way towards the parameters. The first step is to calculate the gradients of the objective function J=L+sJ=L+s with respect to the loss term LL and the regularization term ss.

(4.7.8)

∂J∂L=1and∂J∂s=1.∂J∂L=1and∂J∂s=1.

Next, we compute the gradient of the objective function with respect to variable of the output layer oo according to the chain rule:

(4.7.9)

∂J∂o=prod(∂J∂L,∂L∂o)=∂L∂o∈Rq.∂J∂o=prod(∂J∂L,∂L∂o)=∂L∂o∈Rq.

Next, we calculate the gradients of the regularization term with respect to both parameters:

(4.7.10)

∂s∂W(1)=λW(1)and∂s∂W(2)=λW(2).∂s∂W(1)=λW(1)and∂s∂W(2)=λW(2).

Now we are able to calculate the gradient ∂J/∂W(2)∈Rq×h∂J/∂W(2)∈Rq×h of the model parameters closest to the output layer. Using the chain rule yields:

(4.7.11)

∂J∂W(2)=prod(∂J∂o,∂o∂W(2))+prod(∂J∂s,∂s∂W(2))=∂J∂oh⊤+λW(2).∂J∂W(2)=prod(∂J∂o,∂o∂W(2))+prod(∂J∂s,∂s∂W(2))=∂J∂oh⊤+λW(2).

To obtain the gradient with respect to W(1)W(1) we need to continue backpropagation along the output layer to the hidden layer. The gradient with respect to the hidden layer’s outputs ∂J/∂h∈Rh∂J/∂h∈Rh is given by

(4.7.12)

∂J∂h=prod(∂J∂o,∂o∂h)=W(2)⊤∂J∂o.∂J∂h=prod(∂J∂o,∂o∂h)=W(2)⊤∂J∂o.

Since the activation function ϕϕ applies elementwise, calculating the gradient ∂J/∂z∈Rh∂J/∂z∈Rh of the intermediate variable zz requires that we use the elementwise multiplication operator, which we denote by ⊙⊙:

(4.7.13)

∂J∂z=prod(∂J∂h,∂h∂z)=∂J∂h⊙ϕ′(z).∂J∂z=prod(∂J∂h,∂h∂z)=∂J∂h⊙ϕ′(z).

Finally, we can obtain the gradient ∂J/∂W(1)∈Rh×d∂J/∂W(1)∈Rh×d of the model parameters closest to the input layer. According to the chain rule, we get

(4.7.14)[¶](https://d2l.ai/chapter_multilayer-perceptrons/backprop.html" \l "equation-chapter-multilayer-perceptrons-backprop-11" \o "Permalink to this equation)

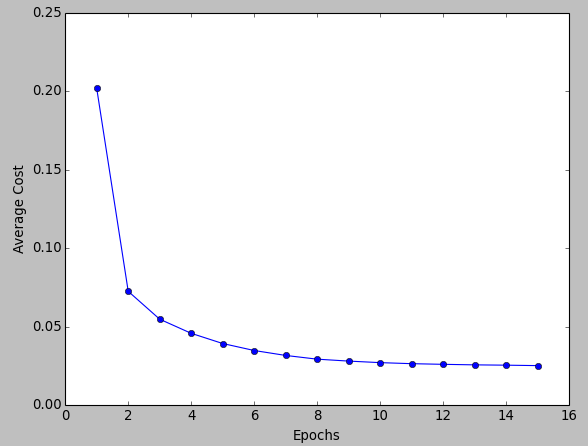
∂J∂W(1)=prod(∂J∂z,∂z∂W(1))+prod(∂J∂s,∂s∂W(1))=∂J∂zx⊤+λW(1).

1. Explain briefly the following variant of Gradient Descent: Stochastic, Batch, and Mini-batch?

# **Batch Gradient Descent**

In Batch Gradient Descent, all the training data is taken into consideration to take a single step. We take the average of the gradients of all the training examples and then use that mean gradient to update our parameters. So that’s just one step of gradient descent in one epoch.

Batch Gradient Descent is great for convex or relatively smooth error manifolds. In this case, we move somewhat directly towards an optimum solution.



Cost vs Epochs (Source: <https://www.bogotobogo.com/python/scikit-learn/scikit-learn_batch-gradient-descent-versus-stochastic-gradient-descent.php>)

The graph of cost vs epochs is also quite smooth because we are averaging over all the gradients of training data for a single step. The cost keeps on decreasing over the epochs.

# **Stochastic Gradient Descent**

In Batch Gradient Descent we were considering all the examples for every step of Gradient Descent. But what if our dataset is very huge. Deep learning models crave for data. The more the data the more chances of a model to be good. Suppose our dataset has 5 million examples, then just to take one step the model will have to calculate the gradients of all the 5 million examples. This does not seem an efficient way. To tackle this problem we have Stochastic Gradient Descent. In Stochastic Gradient Descent (SGD), we consider just one example at a time to take a single step. We do the following steps in **one epoch** for SGD:

1. Take an example
2. Feed it to Neural Network
3. Calculate it’s gradient
4. Use the gradient we calculated in step 3 to update the weights
5. Repeat steps 1–4 for all the examples in training dataset

Since we are considering just one example at a time the cost will fluctuate over the training examples and it will **not**necessarily decrease. But in the long run, you will see the cost decreasing with fluctuations.



Cost vs Epochs in SGD (Source: <https://adventuresinmachinelearning.com/stochastic-gradient-descent/>)

Also because the cost is so fluctuating, it will never reach the minima but it will keep dancing around it.

SGD can be used for larger datasets. It converges faster when the dataset is large as it causes updates to the parameters more frequently.

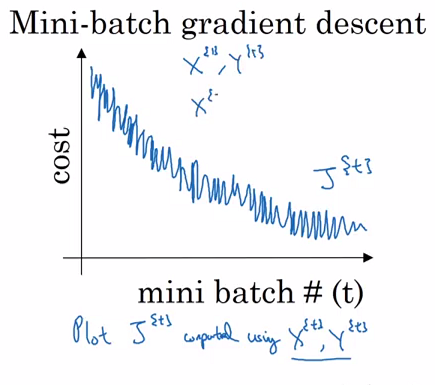
# **Mini Batch Gradient Descent**

We have seen the Batch Gradient Descent. We have also seen the Stochastic Gradient Descent. Batch Gradient Descent can be used for smoother curves. SGD can be used when the dataset is large. Batch Gradient Descent converges directly to minima. SGD converges faster for larger datasets. But, since in SGD we use only one example at a time, we cannot implement the vectorized implementation on it. This can slow down the computations. To tackle this problem, a mixture of Batch Gradient Descent and SGD is used.

Neither we use all the dataset all at once nor we use the single example at a time. We use a batch of a fixed number of training examples which is less than the actual dataset and call it a mini-batch. Doing this helps us achieve the advantages of both the former variants we saw. So, after creating the mini-batches of fixed size, we do the following steps in **one epoch:**

1. Pick a mini-batch
2. Feed it to Neural Network
3. Calculate the mean gradient of the mini-batch
4. Use the mean gradient we calculated in step 3 to update the weights
5. Repeat steps 1–4 for the mini-batches we created

Just like SGD, the average cost over the epochs in mini-batch gradient descent fluctuates because we are averaging a small number of examples at a time.



Cost vs no of mini-batch (Source: <https://stats.stackexchange.com/questions/310734/why-is-the-mini-batch-gradient-descents-cost-function-graph-noisy>)

So, when we are using the mini-batch gradient descent we are updating our parameters frequently as well as we can use vectorized implementation for faster computations.

1. What are the main benefits of Mini-batch Gradient Descent?

Mini-batch SGD is the only way to fit a model into memory. When your dataset is several hundred gigabytes on disk, in compressed form, you simply can't fit them all in memory at once. It's not so much a matter of "pros vs cons", it's a matter of necessity in most tasks (it would be prohibitively expensive to be constantly writing to disk). When you make a CNN model, and you already have the hardware, you don't have any decision of how big your mini-batch is. You just choose the largest mini-batch that fits into memory.

Benefits:

1. High throughput: With mini-batch one can process a large number of input examples per second. The mini batching style of gradient descent is perhaps the only way to use the large number of cores at once in a GPU.
2. (Sometimes) faster convergence: The high througput may also translate to faster convergence depending on the variance in the dataset and the learning rate used.
3. High quality gradient: Mini batching allows for a high quality gradient and this will be really useful allowing one to use high learning rates.

Cons:

1. Low final accuracy: Many times mini batching may result in low final accuracy since the noise in the gradient is really helpful towards the end to extract that last 0.5%.
2. What is transfer learning?

Sophisticated deep learning models have millions of parameters (weights) and training them from scratch often requires large amounts of data of computing resources. Transfer learning is a technique that shortcuts much of this by taking a piece of a model that has already been trained on a related task and reusing it in a new model.

For example, the next tutorial in this section will show you how to build your own image recognizer that takes advantage of a model that was already trained to recognize 1000s of different kinds of objects within images. You can adapt the existing knowledge in the pre-trained model to detect your own image classes using much less training data than the original model required.

This is useful for rapidly developing new models as well as customizing models in resource contstrained environments like browsers and mobile devices.

Most often when doing transfer learning, we don't adjust the weights of the original model. Instead we remove the final layer and train a new (often fairly shallow) model on top of the output of the truncated model. This is the technique you will see demonstrated in the tutorials in this section.