Computer Vision

Lecture 09
Training Neural Networks
Military College of Signals

Asim D. Bakhshi
asim.dilawar@mcs.edu.pk

June 9, 2022

Contents

1 Setting Up the Data Model

In a previous lecture we introduced a model of a Neuron, which computes a dot product following a non-linearity, and Neural Networks that arrange neurons into layers. Together, these choices define the new form of the score function, which we have extended from the simple linear mapping that we have seen in the Linear Classification section. In particular, a Neural Network performs a

sequence of linear mappings with interwoven non-linearities. In this section we will discuss additional design choices regarding data preprocessing, weight initialization, and loss functions.

1.1 Data Preprocessing

There are three common forms of data preprocessing a data matrix X, where we will assume that X is of size $[N \times D]$ (N is the number of data, D is their dimensionality).

1.1.1 Mean Subtraction

Mean subtraction is the most common form of preprocessing. It involves subtracting the mean across every individual feature in the data, and has the geometric interpretation of centering the cloud of data around the origin along every dimension. With images specifically, for convenience it can be common to subtract a single value from all pixels or to do so separately across the three color channels.

1.1.2 Normalization

Normalization refers to normalizing the data dimensions so that they are of approximately the same scale. There are two common ways of achieving this normalization. One is to divide each dimension by its standard deviation, once it has been zero-centered. Another form of this preprocessing normalizes each dimension so that the min and max along the dimension is -1 and 1 respectively. It only makes sense to apply this preprocessing if you have a reason to believe that different input features have different scales (or units), but they should be of approximately equal importance to the learning algorithm. In case of images, the relative scales of pixels are already approximately equal (and in range from 0 to 255), so it is not strictly necessary to perform this additional preprocessing step.

1.1.3 PCA and Whitening

PCA and Whitening is another form of preprocessing. In this process, the data is first centered as described above. Then, we can compute the covariance matrix that tells us about the correlation structure in the data. The (i, j) element of the data covariance matrix contains the covariance between i-th and j-th dimension of the data. In particular, the diagonal of this matrix contains the variances. Furthermore, the covariance matrix is symmetric and positive semi-definite. We can compute the SVD factorization of the data covariance matrix. where the columns of U are the eigenvectors and S is a 1-D array of the singular values. To decorrelate the data, we project the original (but zero-centered) data into the eigenbasis. The columns of U are a set of orthonormal vectors (norm of 1, and orthogonal to each other), so they can be regarded as basis vectors. The projection therefore corresponds to a rotation of the data in X so that the new

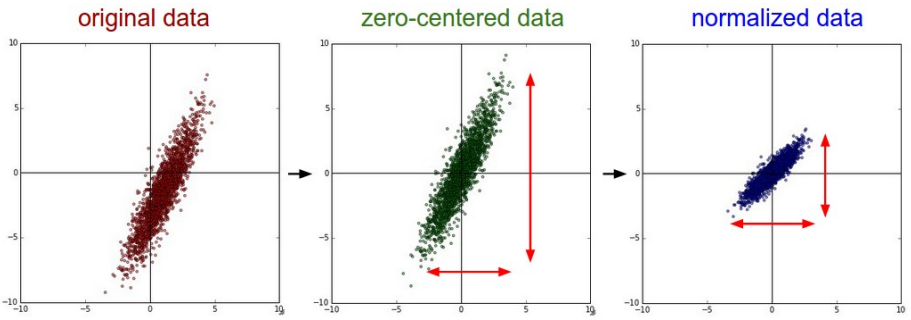


Figure 1: Common data preprocessing pipeline. Left: Original toy, 2-dimensional input data. Middle: The data is zero-centered by subtracting the mean in each dimension. The data cloud is now centered around the origin. Right: Each dimension is additionally scaled by its standard deviation. The red lines indicate the extent of the data - they are of unequal length in the middle, but of equal length on the right.

axes are the eigenvectors. We can use this to reduce the dimensionality of the data by only using the top few eigenvectors, and discarding the dimensions along which the data has no variance. This is also sometimes referred to as Principal Component Analysis (PCA) dimensionality reduction.

After this operation, we would have reduced the original dataset of size $[N \times D]$ to one of size $[N \times 100]$, keeping the 100 dimensions of the data that contain the most variance. It is very often the case that you can get very good performance by training linear classifiers or neural networks on the PCA-reduced datasets, obtaining savings in both space and time.

The last transformation you may see in practice is **whitening**. The whitening operation takes the data in the eigenbasis and divides every dimension by the eigenvalue to normalize the scale. The geometric interpretation of this transformation is that if the input data is a multivariable gaussian, then the whitened data will be a gaussian with zero mean and identity covariance matrix.

Warning: Exaggerating Noise. Note that we're adding 1e-5 (or a small constant) to prevent division by zero. One weakness of this transformation is that it can greatly exaggerate the noise in the data, since it stretches all dimensions (including the irrelevant dimensions of tiny variance that are mostly noise) to be of equal size in the input. This can in practice be mitigated by stronger smoothing (i.e. increasing 1e-5 to be a larger number).

We can also try to visualize these transformations with CIFAR-10 images. The training set of CIFAR-10 is of size 50,000 x 3072, where every image is stretched out into a 3072-dimensional row vector. We can then compute the [3072 x 3072] covariance matrix and compute its SVD decomposition (which can be relatively expensive). What do the computed eigenvectors look like visually?

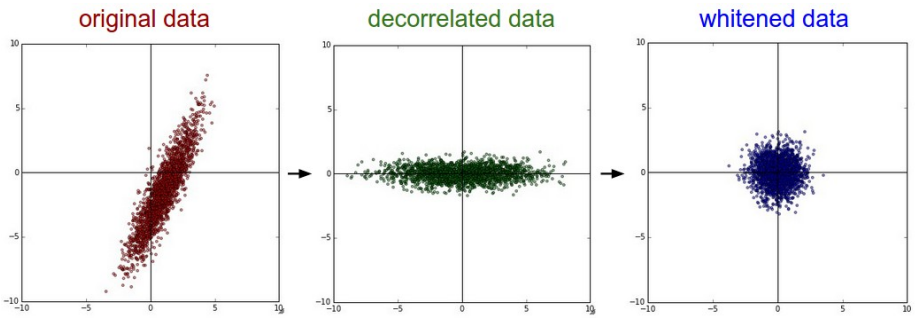


Figure 2: PCA / Whitening. Left: Original toy, 2-dimensional input data. Middle: After performing PCA. The data is centered at zero and then rotated into the eigenbasis of the data covariance matrix. This decorrelates the data (the covariance matrix becomes diagonal). Right: Each dimension is additionally scaled by the eigenvalues, transforming the data covariance matrix into the identity matrix. Geometrically, this corresponds to stretching and squeezing the data into an isotropic gaussian blob.

In Practice. We mention PCA/Whitening in these notes for completeness, but these transformations are not used with Convolutional Networks. However, it is very important to zero-center the data, and it is common to see normalization of every pixel as well.

Common Pitfall. An important point to make about the preprocessing is that any preprocessing statistics (e.g. the data mean) must only be computed on the training data, and then applied to the validation / test data. E.g. computing the mean and subtracting it from every image across the entire dataset and then splitting the data into train/val/test splits would be a mistake. Instead, the mean must be computed only over the training data and then subtracted equally from all splits (train/val/test).

1.2 Weight Initialization

We have seen how to construct a Neural Network architecture, and how to preprocess the data. Before we can begin to train the network we have to initialize its parameters.

1.2.1 Pitfall: All Zero Initialization

Lets start with what we should not do. Note that we do not know what the final value of every weight should be in the trained network, but with proper data normalization it is reasonable to assume that approximately half of the weights will be positive and half of them will be negative. A reasonable-sounding idea then might be to set all the initial weights to zero, which we expect to be

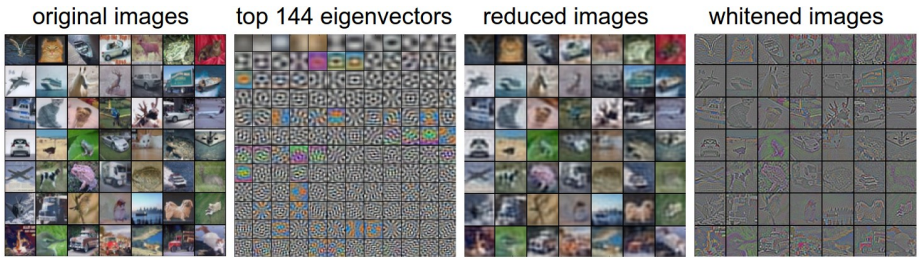


Figure 3: Left: An example set of 49 images. 2nd from Left: The top 144 out of 3072 eigenvectors. The top eigenvectors account for most of the variance in the data, and we can see that they correspond to lower frequencies in the images. 2nd from Right: The 49 images reduced with PCA, using the 144 eigenvectors shown here. That is, instead of expressing every image as a 3072-dimensional vector where each element is the brightness of a particular pixel at some location and channel, every image above is only represented with a 144-dimensional vector, where each element measures how much of each eigenvector adds up to make up the image. In order to visualize what image information has been retained in the 144 numbers, we must rotate back into the "pixel" basis of 3072 numbers. Since U is a rotation, this can be achieved by multiplying by U.transpose()[:144,:], and then visualizing the resulting 3072 numbers as the image. You can see that the images are slightly blurrier, reflecting the fact that the top eigenvectors capture lower frequencies. However, most of the information is still preserved. Right: Visualization of the "white" representation, where the variance along every one of the 144 dimensions is squashed to equal length. Here, the whitened 144 numbers are rotated back to image pixel basis by multiplying by U.transpose()[:144,:]. The lower frequencies (which accounted for most variance) are now negligible, while the higher frequencies (which account for relatively little variance originally) become exaggerated.

the "best guess" in expectation. This turns out to be a mistake, because if every neuron in the network computes the same output, then they will also all compute the same gradients during backpropagation and undergo the exact same parameter updates. In other words, there is no source of asymmetry between neurons if their weights are initialized to be the same.

1.2.2 Small Random Numbers

Therefore, we still want the weights to be very close to zero, but as we have argued above, not identically zero. As a solution, it is common to initialize the weights of the neurons to small numbers and refer to doing so as symmetry breaking. The idea is that the neurons are all random and unique in the beginning, so they will compute distinct updates and integrate themselves as diverse parts of the full network. With this formulation, every neuron's weight vector is initialized as a random vector sampled from a multi-dimensional gaussian, so the neurons point in random direction in the input space. It is also possible to use small numbers drawn from a uniform distribution, but this seems to have relatively little impact on the final performance in practice.

Warning: It's not necessarily the case that smaller numbers will work strictly better. For example, a Neural Network layer that has very small weights will during backpropagation compute very small gradients on its data (since this gradient is proportional to the value of the weights). This could greatly diminish the "gradient signal" flowing backward through a network, and could become a concern for deep networks.

1.2.3 Caliberating the Variances

One problem with the above suggestion is that the distribution of the outputs from a randomly initialized neuron has a variance that grows with the number of inputs. It turns out that we can normalize the variance of each neuron's output to 1 by scaling its weight vector by the square root of its fan-in (i.e. its number of inputs).

1.2.4 Sparse Initialization

Another way to address the uncalibrated variances problem is to set all weight matrices to zero, but to break symmetry every neuron is randomly connected (with weights sampled from a small gaussian as above) to a fixed number of neurons below it. A typical number of neurons to connect to may be as small as 10.

1.2.5 Initializing the Biases

It is possible and common to initialize the biases to be zero, since the asymmetry breaking is provided by the small random numbers in the weights. For ReLU non-linearities, some people like to use small constant value such as 0.01 for all biases because this ensures that all ReLU units fire in the beginning and

therefore obtain and propagate some gradient. However, it is not clear if this provides a consistent improvement (in fact some results seem to indicate that this performs worse) and it is more common to simply use 0 bias initialization.

1.2.6 In Practice

The current recommendation is to use ReLU units and use initialization as discussed in [1].

1.2.7 Batch Normalization

A recently developed technique by Ioffe and Szegedy called Batch Normalization [2] alleviates a lot of headaches with properly initializing neural networks by explicitly forcing the activations throughout a network to take on a unit gaussian distribution at the beginning of the training. The core observation is that this is possible because normalization is a simple differentiable operation. In the implementation, applying this technique usually amounts to insert the BatchNorm layer immediately after fully connected layers (or convolutional layers, as we'll soon see), and before non-linearities. We do not expand on this technique here because it is well described in the linked paper, but note that it has become a very common practice to use Batch Normalization in neural networks. In practice networks that use Batch Normalization are significantly more robust to bad initialization. Additionally, batch normalization can be interpreted as doing preprocessing at every layer of the network, but integrated into the network itself in a differentiable manner.

2 Regularization

There are several ways of controlling the capacity of Neural Networks to prevent overfitting as we have already discussed.

2.1 L2 Regularization

L2 regularization is perhaps the most common form of regularization. It can be implemented by penalizing the squared magnitude of all parameters directly in the objective. That is, for every weight w in the network, we add the term $\frac{1}{2}\lambda w^2$ to the objective, where λ is the regularization strength. It is common to see the factor of $\frac{1}{2}$ in front because then the gradient of this term with respect to the parameter w is simply λw instead of $2\lambda w$. The L2 regularization has the intuitive interpretation of heavily penalizing peaky weight vectors and preferring diffuse weight vectors. As we discussed in the Linear Classification section, due to multiplicative interactions between weights and inputs this has the appealing property of encouraging the network to use all of its inputs a little rather than some of its inputs a lot. Lastly, notice that during gradient descent parameter update, using the L2 regularization ultimately means that every weight is decayed linearly.

2.2 L1 Regularization

L1 regularization is another relatively common form of regularization, where for each weight w we add the term $\lambda \mid w \mid$ to the objective. It is possible to combine the L1 regularization with the L2 regularization: $\lambda_1 \mid w \mid + \lambda_2 w^2$ (this is called Elastic net regularization). The L1 regularization has the intriguing property that it leads the weight vectors to become sparse during optimization (i.e. very close to exactly zero). In other words, neurons with L1 regularization end up using only a sparse subset of their most important inputs and become nearly invariant to the "noisy" inputs. In comparison, final weight vectors from L2 regularization are usually diffuse, small numbers. In practice, if you are not concerned with explicit feature selection, L2 regularization can be expected to give superior performance over L1.

2.3 Max Norm Constraints

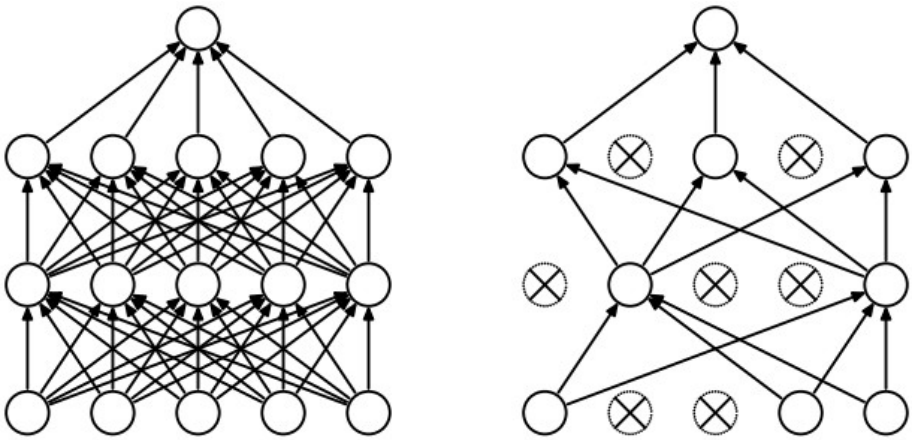
Another form of regularization is to enforce an absolute upper bound on the magnitude of the weight vector for every neuron and use projected gradient descent to enforce the constraint. In practice, this corresponds to performing the parameter update as normal, and then enforcing the constraint by clamping the weight vector $\vec{w}$ of every neuron to satisfy $\|\vec{w}\|_2 < c$. Typical values of c are on orders of 3 or 4. Some people report improvements when using this form of regularization. One of its appealing properties is that network cannot "explode" even when the learning rates are set too high because the updates are always bounded.

2.4 Dropout

Dropout is an extremely effective, simple and recently introduced regularization technique by Srivastava et al [3]. It complements the other methods (L1, L2, maxnorm). While training, dropout is implemented by only keeping a neuron active with some probability p (a hyperparameter), or setting it to zero otherwise.

2.5 Theme of Noise in Forward Pass

Dropout falls into a more general category of methods that introduce stochastic behavior in the forward pass of the network. During testing, the noise is marginalized over analytically (as is the case with dropout when multiplying by p), or numerically (e.g. via sampling, by performing several forward passes with different random decisions and then averaging over them). An example of other research in this direction includes DropConnect [4], where a random set of weights is instead set to zero during forward pass. As foreshadowing, Convolutional Neural Networks also take advantage of this theme with methods such as stochastic pooling, fractional pooling, and data augmentation. We will go into details of these methods later.



(a) Standard Neural Net (b) After applying dropout.

Figure 4: Figure taken from the Dropout paper [3] that illustrates the idea. During training, Dropout can be interpreted as sampling a Neural Network within the full Neural Network, and only updating the parameters of the sampled network based on the input data. (However, the exponential number of possible sampled networks are not independent because they share the parameters.) During testing there is no dropout applied, with the interpretation of evaluating an averaged prediction across the exponentially-sized ensemble of all sub-networks (more about ensembles in the next section).

2.6 Bias Regularization

As we already mentioned in the Linear Classification section, it is not common to regularize the bias parameters because they do not interact with the data through multiplicative interactions, and therefore do not have the interpretation of controlling the influence of a data dimension on the final objective. However, in practical applications (and with proper data preprocessing) regularizing the bias rarely leads to significantly worse performance. This is likely because there are very few bias terms compared to all the weights, so the classifier can "afford to" use the biases if it needs them to obtain a better data loss.

2.7 Per-layer Regularization

It is not very common to regularize different layers to different amounts (except perhaps the output layer). Relatively few results regarding this idea have been published in the literature.

2.8 In Practice

It is most common to use a single, global L2 regularization strength that is cross-validated. It is also common to combine this with dropout applied after all layers. The value of $p = 0.5$ is a reasonable default, but this can be tuned on validation data.

3 Loss Function

We have already discussed the loss function in linear classification context.

4 Training Process

In the previous sections we have discussed the static parts of a Neural Networks: how we can set up the network connectivity, the data, and the loss function. This section is devoted to the dynamics, or in other words, the process of learning the parameters and finding good hyperparameters.

4.1 Gradient Checks

In theory, performing a gradient check is as simple as comparing the analytic gradient to the numerical gradient. In practice, the process is much more involved and error prone. Here are some tips, tricks, and issues to watch out for:

- Use the centered formula.
- Use relative error for the comparison.
- Use double precision.

- Stick around active range of floating point. (read through [5]).
- One source of inaccuracy to be aware of during gradient checking is the problem of kinks. Kinks refer to non-differentiable parts of an objective function, introduced by functions such as ReLU, the SVM loss, Maxout neurons, etc.
- Use only few datapoints.
- Be careful with the step size h.
- Don't let the regularization overwhelm the data.
- Remember to turn off dropout/augmentations.
- In practice the gradients can have sizes of million parameters. In these cases it is only practical to check some of the dimensions of the gradient and assume that the others are correct. Be careful: One issue to be careful with is to make sure to gradient check a few dimensions for every separate parameter. In some applications, people combine the parameters into a single large parameter vector for convenience. In these cases, for example, the biases could only take up a tiny number of parameters from the whole vector, so it is important to not sample at random but to take this into account and check that all parameters receive the correct gradients.

4.2 Sanity Checks Before Learning

Here are a few sanity checks you might consider running before you plunge into expensive optimization:

- Look for correct loss at chance performance. Make sure you're getting the loss you expect when you initialize with small parameters. It's best to first check the data loss alone (so set regularization strength to zero). For example, for CIFAR-10 with a Softmax classifier we would expect the initial loss to be 2.302, because we expect a diffuse probability of 0.1 for each class (since there are 10 classes), and Softmax loss is the negative log probability of the correct class so: $-\ln(0.1) = 2.302$. For SVM, we expect all desired margins to be violated (since all scores are approximately zero), and hence expect a loss of 9 (since margin is 1 for each wrong class). If you're not seeing these losses there might be issue with initialization.
- As a second sanity check, increasing the regularization strength should increase the loss.
- Overfit a tiny subset of data. Lastly and most importantly, before training on the full dataset try to train on a tiny portion (e.g. 20 examples) of your data and make sure you can achieve zero cost. For this experiment it's also best to set regularization to zero, otherwise this can prevent you from getting zero cost. Unless you pass this sanity check with a small

dataset it is not worth proceeding to the full dataset. Note that it may happen that you can overfit very small dataset but still have an incorrect implementation. For instance, if your datapoints' features are random due to some bug, then it will be possible to overfit your small training set but you will never notice any generalization when you fold it your full dataset.

4.3 Babysitting the Learning Process

There are multiple useful quantities you should monitor during training of a neural network. These plots are the window into the training process and should be utilized to get intuitions about different hyperparameter settings and how they should be changed for more efficient learning.

The x-axis of the plots below are always in units of epochs, which measure how many times every example has been seen during training in expectation (e.g. one epoch means that every example has been seen once). It is preferable to track epochs rather than iterations since the number of iterations depends on the arbitrary setting of batch size.

4.3.1 Loss Function

The first quantity that is useful to track during training is the loss, as it is evaluated on the individual batches during the forward pass. Figure 5 shows the loss over time, and especially what the shape might tell you about the learning rate.

The amount of "wiggle" in the loss is related to the batch size. When the batch size is 1, the wiggle will be relatively high. When the batch size is the full dataset, the wiggle will be minimal because every gradient update should be improving the loss function monotonically (unless the learning rate is set too high).

Some people prefer to plot their loss functions in the log domain. Since learning progress generally takes an exponential form shape, the plot appears as a slightly more interpretable straight line, rather than a hockey stick. Additionally, if multiple cross-validated models are plotted on the same loss graph, the differences between them become more apparent.

4.3.2 Train/Val Accuracy

The second important quantity to track while training a classifier is the validation/training accuracy. This plot in Figure 6 can give you valuable insights into the amount of overfitting in your model.

4.3.3 Ratio of Weights:Updates

The last quantity you might want to track is the ratio of the update magnitudes to the value magnitudes. Note: updates, not the raw gradients (e.g. in vanilla sgd this would be the gradient multiplied by the learning rate). You might want to evaluate and track this ratio for every set of parameters independently. A

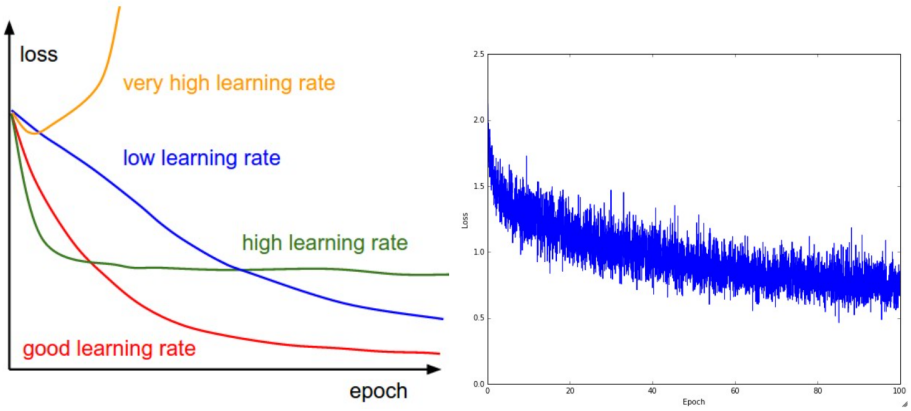


Figure 5: Left: A figure depicting the effects of different learning rates. With low learning rates the improvements will be linear. With high learning rates they will start to look more exponential. Higher learning rates will decay the loss faster, but they get stuck at worse values of loss (green line). This is because there is too much "energy" in the optimization and the parameters are bouncing around chaotically, unable to settle in a nice spot in the optimization landscape. Right: An example of a typical loss function over time, while training a small network on CIFAR-10 dataset. This loss function looks reasonable (it might indicate a slightly too small learning rate based on its speed of decay, but it's hard to say), and also indicates that the batch size might be a little too low (since the cost is a little too noisy).

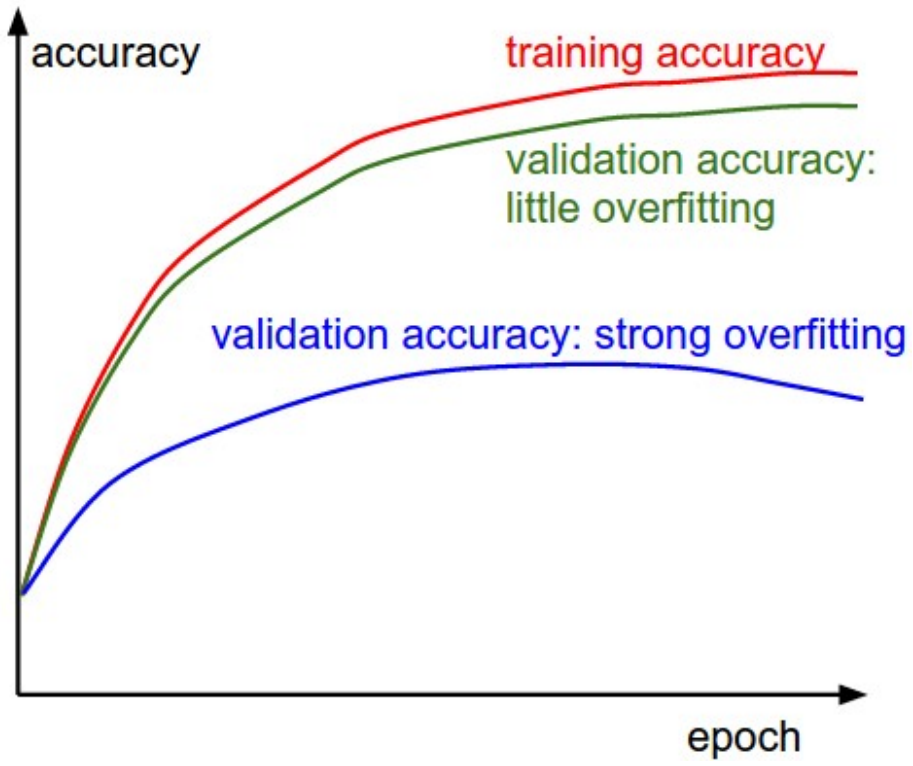


Figure 6: The gap between the training and validation accuracy indicates the amount of overfitting. Two possible cases are shown in the diagram on the left. The blue validation error curve shows very small validation accuracy compared to the training accuracy, indicating strong overfitting (note, it's possible for the validation accuracy to even start to go down after some point). When you see this in practice you probably want to increase regularization (stronger L2 weight penalty, more dropout, etc.) or collect more data. The other possible case is when the validation accuracy tracks the training accuracy fairly well. This case indicates that your model capacity is not high enough: make the model larger by increasing the number of parameters.

rough heuristic is that this ratio should be somewhere around $1e-3$. If it is lower than this then the learning rate might be too low. If it is higher then the learning rate is likely too high.

Instead of tracking the min or the max, some people prefer to compute and track the norm of the gradients and their updates instead. These metrics are usually correlated and often give approximately the same results.

4.3.4 Activation/ Gradient Distributions Per layer

An incorrect initialization can slow down or even completely stall the learning process. Luckily, this issue can be diagnosed relatively easily. One way to do so is to plot activation/gradient histograms for all layers of the network. Intuitively, it is not a good sign to see any strange distributions - e.g. with tanh neurons we would like to see a distribution of neuron activations between the full range of $[-1, 1]$, instead of seeing all neurons outputting zero, or all neurons being completely saturated at either -1 or 1.

4.3.5 First-layer Visualizations

Lastly, when one is working with image pixels it can be helpful and satisfying to plot the first-layer features visually. An example is shown in Figure 7.

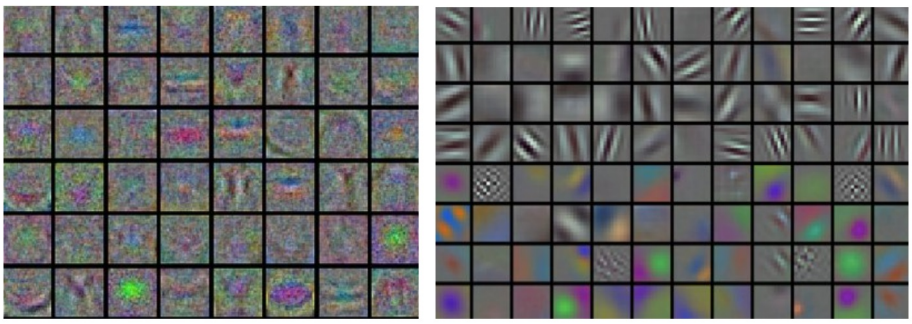


Figure 7: Examples of visualized weights for the first layer of a neural network. Left: Noisy features indicate could be a symptom: Unconverged network, improperly set learning rate, very low weight regularization penalty. Right: Nice, smooth, clean and diverse features are a good indication that the training is proceeding well.

4.4 Parameter Updates

4.4.1 SGD and Variants

Vanilla Update. The simplest form of update is to change the parameters along the negative gradient direction (since the gradient indicates the direction of increase, but we usually wish to minimize a loss function).

Momentum Update. Another approach that almost always enjoys better converge rates on deep networks. This update can be motivated from a physical perspective of the optimization problem. In particular, the loss can be interpreted as the height of a hilly terrain (and therefore also to the potential energy since $U = mgh$ and therefore $U \propto h$). Initializing the parameters with random numbers is equivalent to setting a particle with zero initial velocity at some location. The optimization process can then be seen as equivalent to the process of simulating the parameter vector (i.e. a particle) as rolling on the landscape.

Since the force on the particle is related to the gradient of potential energy (i.e. $F = -\nabla U$), the force felt by the particle is precisely the (negative) gradient of the loss function. Moreover, $F = ma$ so the (negative) gradient is in this view proportional to the acceleration of the particle. Note that this is different from the SGD update shown above, where the gradient directly integrates the position. Instead, the physics view suggests an update in which the gradient only directly influences the velocity, which in turn has an effect on the position.

Nesterov Momentum. This is a slightly different version of the momentum update that has recently been gaining popularity. It enjoys stronger theoretical converge guarantees for convex functions and in practice it also consistently works slightly better than standard momentum.

The core idea behind Nesterov momentum is that when the current parameter vector is at some position x, then looking at the momentum update above, we know that the momentum term alone (i.e. ignoring the second term with the gradient) is about to nudge the parameter vector by μv. Therefore, if we are about to compute the gradient, we can treat the future approximate position $x + \mu v$ as a "lookahead" - this is a point in the vicinity of where we are soon going to end up. Hence, it makes sense to compute the gradient at $x + \mu v$ instead of at the "old/stale" position x.

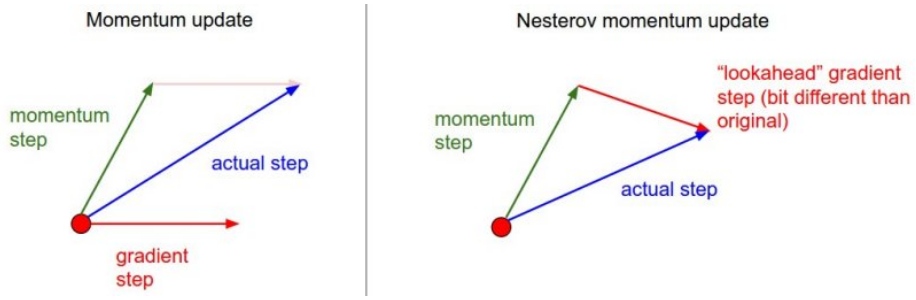


Figure 8: Nesterov momentum. Instead of evaluating gradient at the current position (red circle), we know that our momentum is about to carry us to the tip of the green arrow. With Nesterov momentum we therefore instead evaluate the gradient at this "looked-ahead" position.

4.4.2 Annealing the Learning Rate

In training deep networks, it is usually helpful to anneal the learning rate over time. Good intuition to have in mind is that with a high learning rate, the system contains too much kinetic energy and the parameter vector bounces around chaotically, unable to settle down into deeper, but narrower parts of the loss function. Knowing when to decay the learning rate can be tricky: Decay it slowly and you'll be wasting computation bouncing around chaotically with little improvement for a long time. But decay it too aggressively and the system will cool too quickly, unable to reach the best position it can. There are three common types of implementing the learning rate decay:

- **Step decay:** Reduce the learning rate by some factor every few epochs. Typical values might be reducing the learning rate by a half every 5 epochs, or by 0.1 every 20 epochs. These numbers depend heavily on the type of problem and the model. One heuristic you may see in practice is to watch the validation error while training with a fixed learning rate, and reduce the learning rate by a constant (e.g. 0.5) whenever the validation error stops improving.
- **Exponential Decay** has the mathematical form $\alpha = \alpha_0 e^{-kt}$, where α_0, k are hyperparameters and t is the iteration number (but you can also use units of epochs).
- **1/t Decay** has the mathematical form $\alpha = \alpha_0/(1 + kt)$ where a_0, k are hyperparameters and t is the iteration number.

In practice, we find that the step decay is slightly preferable because the hyperparameters it involves (the fraction of decay and the step timings in units of epochs) are more interpretable than the hyperparameter k. Lastly, if you can afford the computational budget, err on the side of slower decay and train for a longer time.

4.4.3 Per-parameter Adaptive Learning Rate Methods

All previous approaches we've discussed so far manipulated the learning rate globally and equally for all parameters. Tuning the learning rates is an expensive process, so much work has gone into devising methods that can adaptively tune the learning rates, and even do so per parameter. Many of these methods may still require other hyperparameter settings, but the argument is that they are well-behaved for a broader range of hyperparameter values than the raw learning rate. In this section we highlight some common adaptive methods you may encounter in practice:

Adagrad is an adaptive learning rate method originally proposed by [6].

RMSprop is a very effective adaptive learning rate method. The RMSProp update adjusts the Adagrad method in a very simple way in an attempt to reduce its aggressive, monotonically decreasing learning rate. In particular, it uses a moving average of squared gradients instead.

Adam is an update method that looks a bit like RMSProp with momentum.

4.5 Hyperparameter Optimization

As we have seen, training Neural Networks can involve many hyperparameter settings. The most common hyperparameters in context of Neural Networks include:

- the initial learning rate
- learning rate decay schedule (such as the decay constant)
- regularization strength (L2 penalty, dropout strength)

But as we saw, there are many more relatively less sensitive hyperparameters, for example in per-parameter adaptive learning methods, the setting of momentum and its schedule, etc. In this section we describe some additional tips and tricks for performing the hyperparameter search.

4.5.1 Implementation

Larger Neural Networks typically require a long time to train, so performing hyperparameter search can take many days/weeks. It is important to keep this in mind since it influences the design of your code base. One particular design is to have a worker that continuously samples random hyperparameters and performs the optimization. During the training, the worker will keep track of the validation performance after every epoch, and writes a model checkpoint (together with miscellaneous training statistics such as the loss over time) to a file, preferably on a shared file system. It is useful to include the validation performance directly in the filename, so that it is simple to inspect and sort the progress. Then there is a second program which we will call a master, which launches or kills workers across a computing cluster, and may additionally inspect the checkpoints written by workers and plot their training statistics, etc.

4.5.2 Prefer one Validation fold to Cross-Validation

In most cases a single validation set of respectable size substantially simplifies the code base, without the need for cross-validation with multiple folds. You'll hear people say they "cross-validated" a parameter, but many times it is assumed that they still only used a single validation set.

4.5.3 Hyperparameter Ranges

Search for hyperparameters on log scale. For example, a typical sampling of the learning rate would generate a random number from a uniform distribution and raise it to the power of 10. The same strategy should be used for the regularization strength. Intuitively, this is because learning rate and regularization strength have multiplicative effects on the training dynamics. For example, a fixed change of adding 0.01 to a learning rate has huge effects on the dynamics if the learning rate is 0.001, but nearly no effect if the learning rate when it is 10. This is because the learning rate multiplies the computed gradient in the update. Therefore, it is much more natural to consider a range of learning rate multiplied or divided by some value, than a range of learning rate added or subtracted to by some value.

4.5.4 Prefer Random Search to Grid Search

As argued by Bergstra and Bengio in [7], "randomly chosen trials are more efficient for hyper-parameter optimization than trials on a grid". As it turns out, this is also usually easier to implement.

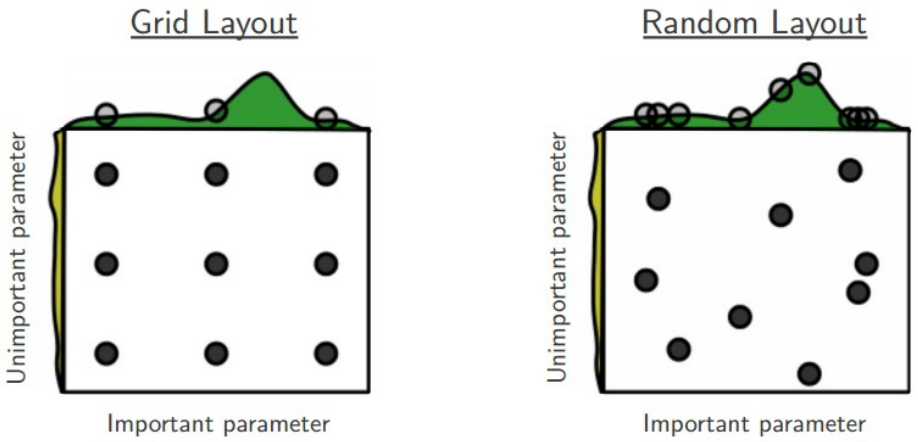


Figure 9: Core illustration from [7]. It is very often the case that some of the hyperparameters matter much more than others (e.g. top hyperparam vs. left one in this figure). Performing random search rather than grid search allows you to much more precisely discover good values for the important ones.

4.5.5 Careful with Best Values on Border

Sometimes it can happen that you're searching for a hyperparameter (e.g. learning rate) in a bad range. Once we receive the results, it is important to double check that the final learning rate is not at the edge of this interval, or otherwise you may be missing more optimal hyperparameter setting beyond the interval.

4.5.6 Stage Your Search from Coarse to Fine

In practice, it can be helpful to first search in coarse ranges, and then depending on where the best results are turning up, narrow the range. Also, it can be helpful to perform the initial coarse search while only training for 1 epoch or even less, because many hyperparameter settings can lead the model to not learn at all, or immediately explode with infinite cost. The second stage could then perform a narrower search with 5 epochs, and the last stage could perform a detailed search in the final range for many more epochs (for example).

4.5.7 Bayesian Hyperparameter Optimization

s a whole area of research devoted to coming up with algorithms that try to more efficiently navigate the space of hyperparameters. The core idea is to appropriately balance the exploration - exploitation trade-off when querying the performance at different hyperparameters. Multiple libraries have been developed based on these models as well.

5 Evaluation

5.1 Model Ensembles

In practice, one reliable approach to improving the performance of Neural Networks by a few percent is to train multiple independent models, and at test time average their predictions. As the number of models in the ensemble increases, the performance typically monotonically improves (though with diminishing returns). Moreover, the improvements are more dramatic with higher model variety in the ensemble. There are a few approaches to forming an ensemble:

- **Same model, different initializations.** Use cross-validation to determine the best hyperparameters, then train multiple models with the best set of hyperparameters but with different random initialization. The danger with this approach is that the variety is only due to initialization.
- **Top models discovered during cross-validation.** Use cross-validation to determine the best hyperparameters, then pick the top few (e.g. 10) models to form the ensemble. This improves the variety of the ensemble but has the danger of including suboptimal models. In practice, this can be easier to perform since it doesn't require additional retraining of models after cross-validation.
- **Different checkpoints of a single model.** If training is very expensive, some people have had limited success in taking different checkpoints of a single network over time (for example after every epoch) and using those to form an ensemble. Clearly, this suffers from some lack of variety, but can still work reasonably well in practice. The advantage of this approach is that is very cheap.

- **Running average of parameters during training.** Related to the last point, a cheap way of almost always getting an extra percent or two of performance is to maintain a second copy of the network's weights in memory that maintains an exponentially decaying sum of previous weights during training. This way you're averaging the state of the network over last several iterations. You will find that this "smoothed" version of the weights over last few steps almost always achieves better validation error. The rough intuition to have in mind is that the objective is bowl-shaped and your network is jumping around the mode, so the average has a higher chance of being somewhere nearer the mode.

One disadvantage of model ensembles is that they take longer to evaluate on test example. An interested reader may find Geoff Hinton's work on "Dark Knowledge" inspiring [8], where the idea is to "distill" a good ensemble back to a single model by incorporating the ensemble log likelihoods into a modified objective.

6 Summary

To train a Neural Network:

- Gradient check your implementation with a small batch of data and be aware of the pitfalls.
- As a sanity check, make sure your initial loss is reasonable, and that you can achieve 100% training accuracy on a very small portion of the data.
- During training, monitor the loss, the training/validation accuracy, and if you're feeling fancier, the magnitude of updates in relation to parameter values (it should be 1e-3), and when dealing with ConvNets, the first-layer weights.
- The two recommended updates to use are either SGD+Nesterov Momentum or Adam.
- Decay your learning rate over the period of the training. For example, halve the learning rate after a fixed number of epochs, or whenever the validation accuracy tops off.
- Search for good hyperparameters with random search (not grid search). Stage your search from coarse (wide hyperparameter ranges, training only for 1-5 epochs), to fine (narrower rangers, training for many more epochs).
- Form model ensembles for extra performance.

References

[1] Kaiming He et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification". In: *Proceedings of the IEEE international conference on computer vision*. 2015. Pp. 1026–1034.

[2] Sergey Ioffe and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *International conference on machine learning*. PMLR. 2015. Pp. 448–456.

[3] Nitish Srivastava et al. "Dropout: a simple way to prevent neural networks from overfitting". *The journal of machine learning research* 15, pp. 1929–1958, 2014.

[4] Li Wan et al. "Regularization of neural networks using dropconnect". In: *International conference on machine learning*. PMLR. 2013. Pp. 1058–1066.

[5] David Goldberg. "What every computer scientist should know about floating-point arithmetic". *ACM computing surveys (CSUR)* 23, pp. 5–48, 1991.

[6] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive subgradient methods for online learning and stochastic optimization." *Journal of machine learning research* 12, 2011.

[7] James Bergstra and Yoshua Bengio. "Random search for hyper-parameter optimization." *Journal of machine learning research* 13, 2012.

[8] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. "Dark knowledge". *Presented as the keynote in BayLearn* 2, p. 2, 2014.