CIS 522 - University Of Pennsylvania, Spring 2020

Lecture 5: Components of Deep Learning - Part 2

30 January 2020

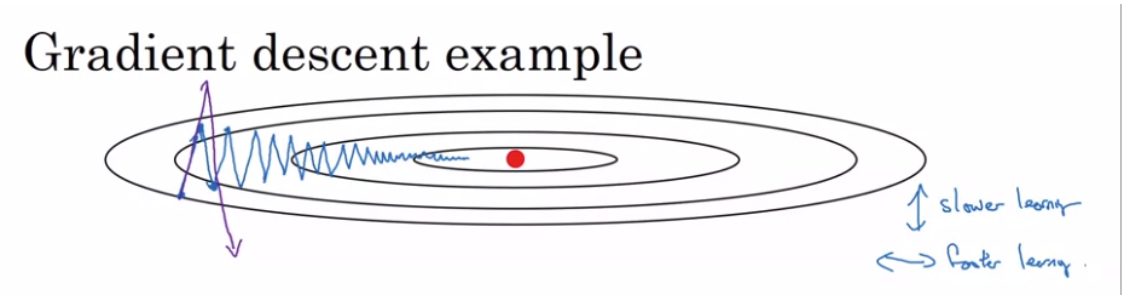
Lecturer: Konrad Kording *Scribe: Kushagra, Jianqiao, Nidhi*

1 Optimization

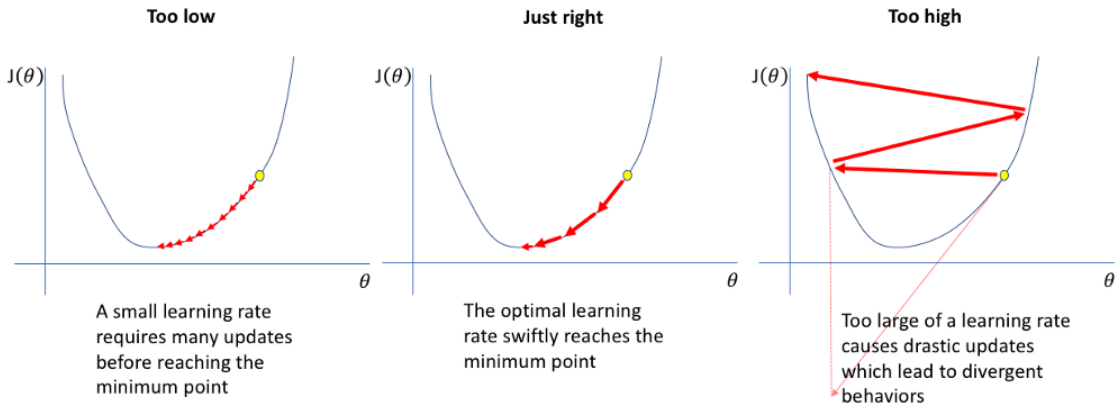
Training a model is to find a minimum point of any model to fit a training dataset, and statistical learning theory implies that a simple model with a smaller VC complexity. In contrast to statistical machine learning algorithms where researchers favor complex and sophisticated optimization algorithms, since the geometry of loss landscape has good structure that we could exploit, however for deep neural network, the loss landscape is much more complex. The theoretical understanding of optimization is an open and popular research topic currently.

There are many types of optimization problems in deep learning, the major one being training a neural network. We define the parameter using θ and the cost function $J(\theta)$. The expected generalization error(risk) is taken over the true data generating distribution p_{data}. If we do have that, it becomes an optimization problem.

$$J(\theta) = \mathbb{E}_{(x,y)\sim p_{data}} \mathcal{L}(f(x;), y) \tag{1}$$



(a) trajector of sgd optimization



(b) trajector of sgd optimization

In the most intuitive method in optimization literature, we need to calculate the exact gradient based on use the whole batch, that is sum over all training examples for the case that involves a finite sum formulation on the training set. As for a very large scale dataset, we usually estimate of the cost function on a subset of the data of the full cost function. The standard error of the mean estimated from n samples is given by $\frac{\sigma}{\sqrt{n}}$, so compare two cases one if we compute the gradient using 100 samples and other using $10,000$ samples then the latter requires 100 times more computation but the former reduces the standard error of mean only by a factor of 10. The algorithms converge much faster if they are allowed to rapidly compute approximate gradients rather than slowly computing exact gradients.

We use the following notations to represent the gradient estimation and objective function:

$$g_{t,i} = \nabla_\theta J(\theta_{t,i}). \tag{2}$$

Gradient-based optimization algorithms update for every parameter θ_t at each time step t then becomes:

$$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i} \tag{3}$$

$G \in \mathbb{R}^{d \times d}$ here is a diagonal matrix where each diagonal element i, i is the sum of the squares of the gradients w.r.t. θ_t up to time step while ϵ is a smoothing term that avoids division by zero.

1.1 Adagrad

The AdaGrad algorithm has a simple and reasonable feature: it assigns different learning rates to each element of the model parameter, namely, the learning rates of all model parameters are scaled inversely proportional to the square root of the sum of all the historical squared values of the gradient [1]. The parameters with the largest partial derivative of the loss have a correspondingly rapid decrease in their learning rate, while parameters with small partial derivatives have a relatively small decrease in their learning rate.

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i} \tag{4}$$

1.2 Adam

Adam[4] is also an adaptive learning rate optimization algorithm where the name "Adam" derives from the phrase "adaptive moments."

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1 - \beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \end{aligned} \tag{5}$$

They counteract these biases by computing bias-corrected first and second moment estimates:

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1 - \beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1 - \beta_2^t} \end{aligned} \tag{6}$$

First, in Adam, momentum is incorporated directly as an estimate of the first-order moment (with exponential weighting) of the gradient. The most straightforward way to add momentum to RMSProp is to apply momentum to the rescaled gradients. The use of momentum in combination with rescaling does not have a clear theoretical motivation. Second, Adam includes bias corrections to the estimates of both the first-order moments (the momentum term) and the (uncentered) second-order moments to account for their initialization at the origin.

1.3 RMSProp

RMSprop is an unpublished, adaptive learning rate method proposed by Geoff Hinton. RMSProp incorporates an estimate of second-order moment; however, it lacks the correction factor. Thus,unlike in Adam, the RMSProp second-order moment estimate may have high bias early in training. Adam is generally regarded as being fairly robust to the choice of hyperparameters, though the learning rate sometimes needs to be changed from the suggested default.8.5.4

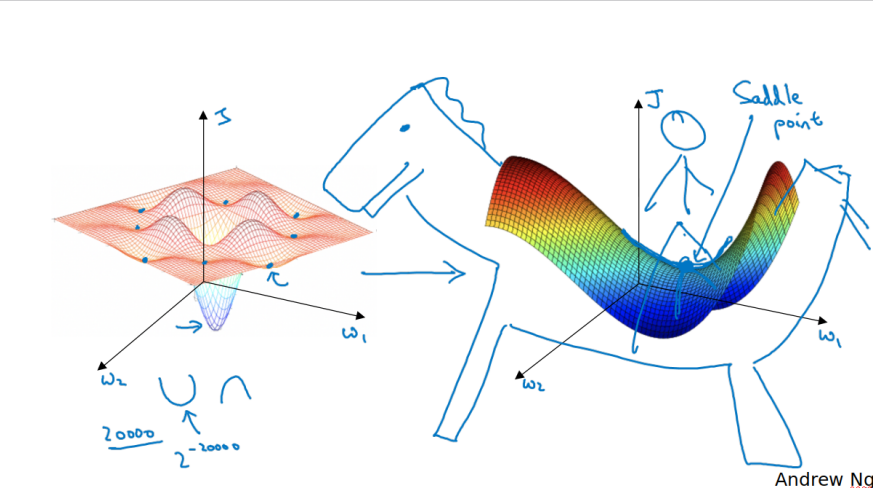
$$\begin{aligned} E[g^2]_t &= 0.9E[g^2]_{t-1} + 0.1g_t^2 \\ \theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t \end{aligned} \tag{7}$$

RMSprop normalizes the learning rate by a factor which is the exponentially decaying average of squared gradients in previous iterations. Hinton suggests γ to be set to 0.9, while a good default value for the learning rate η is 0.001.

At this point, a natural question is: which algorithm should one choose? Unfortunately, there is currently no consensus on this point. Schaul et al. (2014)[5] made an comparison of a large number of optimization algorithms across a wide range of learning tasks. While the results suggest that the family of algorithms with adaptive learning rates.

1.4 Local Minima

In case of convex optimization we could reduce it to the problem of finding a local minima and that local minima was guaranteed to be a global minima as well. Some convex functions have a flat region at the bottom rather than a single global minimum point but any point in that region is acceptable solution. Thus, when optimizing convex functions we know that we have a good solution of any kind of critical point is achieved. With non convex functions we can have many local minima, but we will see it is not necessarily a problem.



(a) saddle point of a loss function

1.5 Improving Adam

Despite the apparent supremacy of adaptive learning rate methods such as Adam, state-of-the-art results for many tasks in computer vision and NLP such as object recognition or machine translation have still been achieved by plain old SGD with momentum. Recent theory [7] provides some justification for this, suggesting that adaptive learning rate methods converge to different (and less optimal) minima than SGD with momentum. It is empirically shown that the minima found by adaptive learning rate methods perform generally worse compared to those found by SGD with momentum on object recognition, character-level language modeling, and constituency parsing. This seems counter-intuitive given that Adam comes with nice convergence guarantees and that its adaptive learning rate should give it an edge over the regular SGD. However, Adam and other adaptive learning rate methods are not without their own flaws.

One factor that partially accounts for Adam's poor generalization ability compared with SGD with momentum on some datasets is weight decay. Weight decay is most commonly used in image classification problems and decays the weights θ_t after every parameter update by multiplying them by a decay rate

$$\theta_{t+1} = w_t \theta_t \tag{8}$$

that is slightly less than 1

This prevents the weights from growing too large. As such, weight decay can also be understood as an ℓ_2 regularization term that depends on the weight decay rate w_t added to the loss:

$$\mathcal{L} = \frac{w_t}{2} \|\theta\|_2^2 \tag{9}$$

Weight decay is commonly implemented in many neural network libraries either as the above regularization term or directly to modify the gradient. As the gradient is modified in both the momentum and Adam update equations (via multiplication with other decay terms), weight decay no longer equals ℓ_2 regularization.

2 Regularization

2.1 Parametric Regularization

In parametric regularization, instead of just finding the θ that minimizes the loss function $L(\theta)$, we usually minimize a composite objective function that is a sum of two terms: $L(\theta) + \lambda R(\theta)$, where the second term $\lambda R(\theta)$ is called a regularization term. Here, $R(\theta)$ is a scalar function of θ and λ is a positive real number. This additional term usually biases the optimal θ's away from minimizing $L(\theta)$, but endowing the optimal θ with useful properties, as described below.

2.1.1 L2 Regularization

The most common regularization is the L2 regularization, in which $R(\theta)$ is the square of the ℓ_2 norm of θ. That is, if $\theta = [\theta_1, \theta_2, \ldots \theta_n]$,

$$R(\theta) = \|\theta\|_2^2 = \left(\theta_1^2 + \theta_2^2 + \ldots \theta_n^2\right).$$

By including this term in the objective, the optimal θ gets biased towards smaller values. Specifically, when $\lambda \to \infty$, $\theta \to 0$, entirely ignoring the loss function. The scaling factor λ allows us to tune the importance of this regularization term relative to the loss function.

By biasing θ toward smaller values, we implicitly constrain the space of functions the resulting network can approximate, usually picking out smoother functions that change less rapidly with the inputs. Perhaps because real world functions are not arbitrarily complex, such regularization may promote better generalization and prevents over-fitting.

2.1.2 L1 Regularization

The ℓ_2 norm used in L2 regularization is just the most familiar special case of the general ℓ_p norm, which is defined as

$$\|\theta\|_p = \left(\theta_1^p + \theta_2^p + \ldots \theta_n^p\right)^{(1/p)}.$$

Another useful and common special case is using the ℓ_1 norm as a regularization term:

$$R(\theta) = \|\theta\|_1 = |\theta_1| + |\theta_2| + \ldots |\theta_n|.$$

Again, increasing λ biases θ toward zero, but with one important difference: the resulting θ tends to be sparse, that is, consisting of lots of zeros. Using a $\|\theta\|_1$ makes the optimization minimize, or at least reduce, the number of non-zero elements, as if it was doing L0 Regularization as below.

2.1.3 L0 Regularization

The ℓ_0 norm is not a true norm (in that it does not satisfy all the mathematical properties that norms should obey) and is simply defined as the number of non-zero elements in θ. Thus, adding the ℓ_0 norm as a regularizer explicitly tries to minimize the number of non-zero elements. Of course, ℓ_0 norm is a highly non-smooth function of θ and does not lend itself to direct optimization. But, it turns out that using the ℓ_1 is a good proxy for using the ℓ_0 norm and does promote sparsity in the optimal solution, as noted in the previous paragraph.

2.2 Early Stopping

Early stopping is a form of regularization used to avoid overfitting when training a learner with an iterative method, such as gradient descent. It does exactly what it sounds like, i.e. stop the training process when the difference between the expected risk of the sample iteration and the minimum expected risk is minimum. In other words use the learned parameters for the step after which the (cross)validation error increases.

2.3 Squeezenet

For a given accuracy level, it is typically possible to identify multiple CNN architectures that achieve that accuracy level. With equivalent accuracy, smaller CNN architectures offer at least three advantages:

- Smaller CNNs require less communication across servers during distributed training.
- Smaller CNNs require less bandwidth to export a new model from the cloud to an autonomous car.
- Smaller CNNs are more feasible to deploy on FPGAs and other hardware with limited memory.

Squeeznet [3] achieve AlexNet-level accuracy on ImageNet with 50x fewer parameters and with compression, Squeezenet comes out to have 510 times smaller size than AlexNet. SqueezeNet employs number of design strategies as listed below to reduce the size of CNN architectures.

1. Replace 3X3 filters with 1X1 filters.
2. Decrease the number of input channels to 3X3 filters.
3. Downsample late in the network so that convolution layers have large activation maps.

It also introduces a Fire Module which comprises of a squeeze convolution layer (which has only 1x1 filters), feeding into an expand layer that has a mix of 1x1 and 3x3 convolution filters as illustrated in Figure 3.

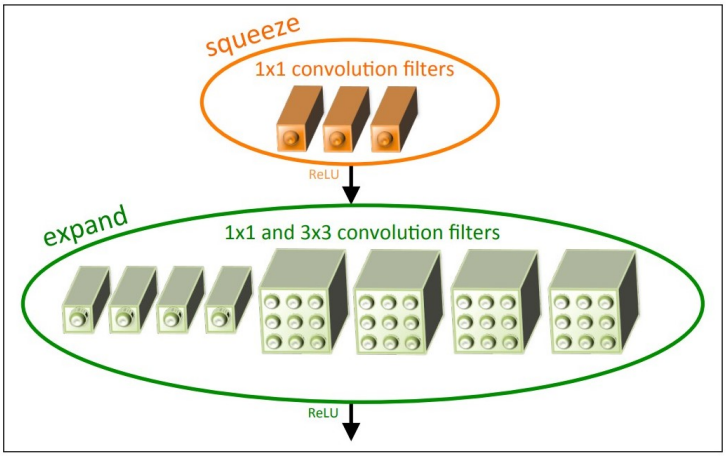


Figure 3: Microarchitectural view: Organization of convolution filters in the Fire module. In this example, $s_{1x1} = 3, e_{1x1} = 4$, and $e_{3x3} = 4$. We illustrate the convolution filters but not the activations.

Three tunable dimensions are exposed (hyperparameters) in a Fire module: s_{1x1}, e_{1x1}, and e_{3x3}. In a Fire module, s1x1 is the number of filters in the squeeze layer (all 1x1), $_{e1x1}$ is the number of 1x1 filters in the expand layer, and e_{3x3} is the number of 3x3 filters in the expand layer. By setting s_{1x1} to be less than $(e_{1x1} + e_{3x3})$, so the squeeze layer helps to limit the number of input channels to the 3x3 filters. Thus the final architecture looks like:

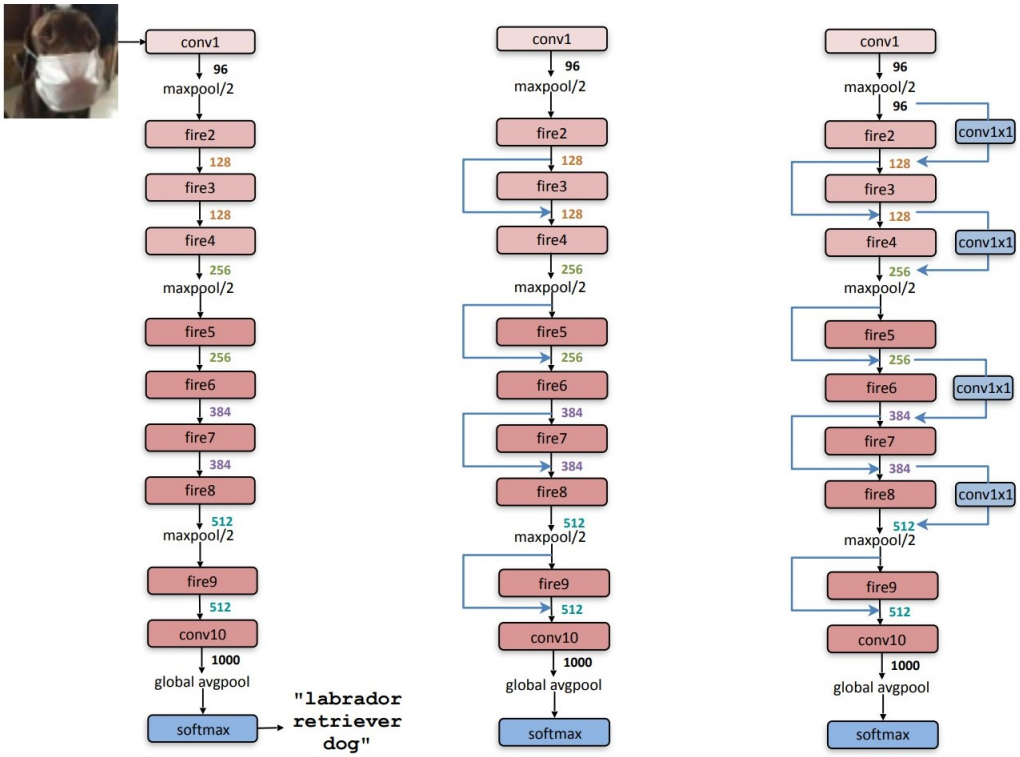


Figure 4: Different Squeezenet Architectures

2.4 Cut Out

Cutout is a simple regularization technique used for convolutional Neural Networks in which the input data is augmented by generating new samples by randomly masking out fixed size square regions of input during the training.

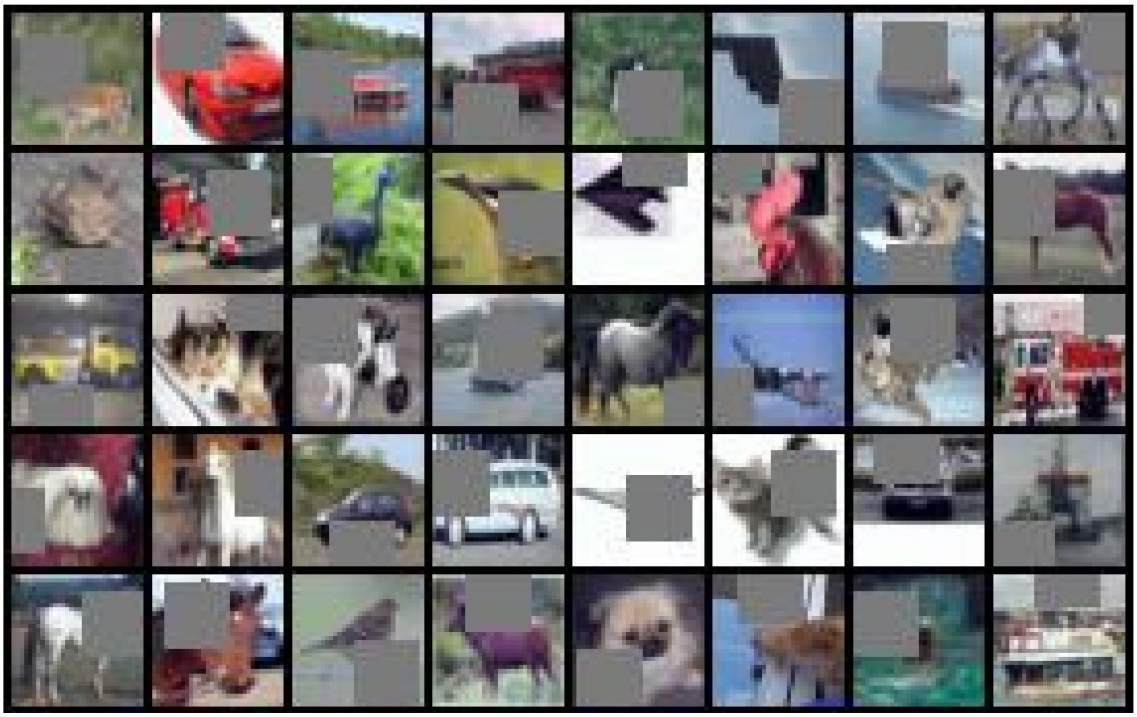


Figure 5: Cutout applied to images from the CIFAR-10 dataset.

2.5 Mixup

Mixup is the technique of augmenting the training data by training a neural network on convex combinations of pairs of examples and their labels. By doing so, mixup[8] regularizes the neural network to favor simple linear behavior in-between training examples, improves the generalization of state-of-the-art neural network architectures, reduces the memorization of corrupt labels, increases the robustness to adversarial examples, and stabilizes the training of generative adversarial networks. In nutshell, mixup constructs virtual training examples

$$\tilde{x} = \lambda x_i + (1-\lambda)x_j; \qquad \text{where } x_i, x_j \text{ are raw input vectors} \tag{10}$$

$$\tilde{y} = \lambda y_i + (1-\lambda)y_j; \qquad \text{where } y_i, y_j \text{ are one-hot label encodings} \tag{11}$$

(x_i, y_i) and (x_j, y_j) are two examples drawn at random from the training data and $\lambda \in [0, 1]$. Despite its simplicity, mixup allows a new state-of-the-art performance in the CIFAR-10, CIFAR100, and ImageNet-2012 image classification datasets.

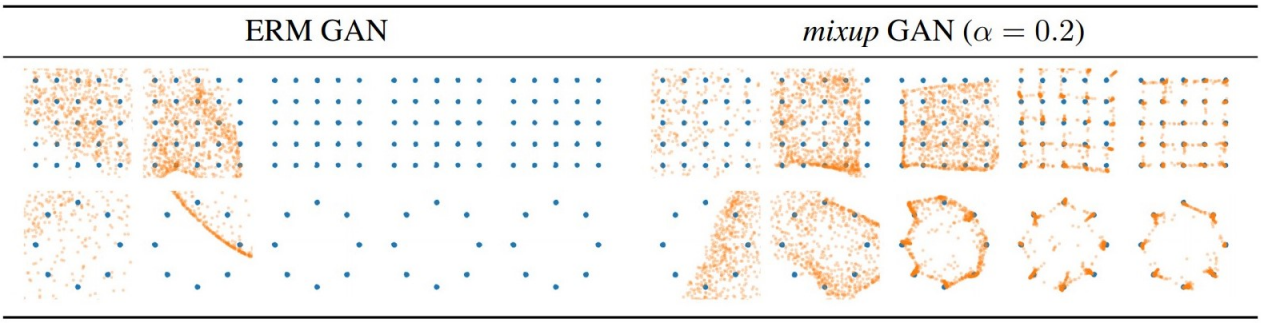


Figure 6: Effect of *mixup* on stabilizing GAN training at iterations 10,100,1000,10000 and 20000.

2.6 Snapshot Ensembling

Snapshot Ensembling [2] produces an ensemble of accurate and diverse models from a single training process. At the heart of Snapshot Ensembling is an optimization process which visits several local minima before converging to a final solution. Snapshot Ensembling take model snapshots at these various minima, and average their predictions at test time. Ensembles work best if the individual models (1) have low test error and (2) do not overlap in the set of examples they misclassify.

- **Cyclic Cosine Annealing**
  To converge to multiple local minima, we follow a cyclic annealing schedule as proposed by Loshchilov Hutter (2016). The learning rate is lowered at a very fast pace, encouraging the model to converge towards its first local minimum after as few as 50 epochs. The optimization is then continued at a larger learning rate, which perturbs the model and dislodges it from the minimum. This process is repeated several times to obtain multiple convergences.

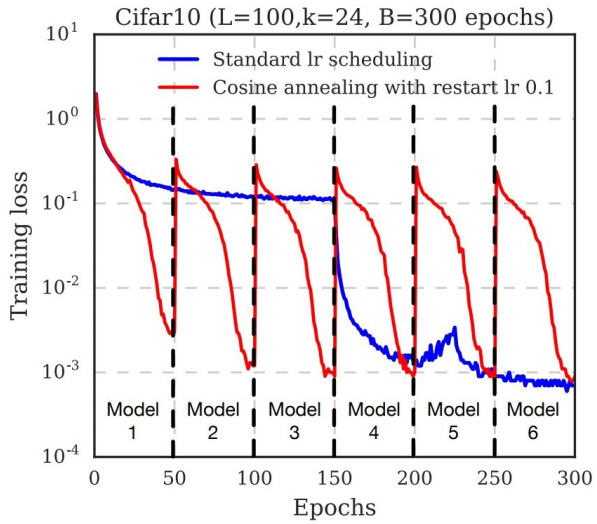


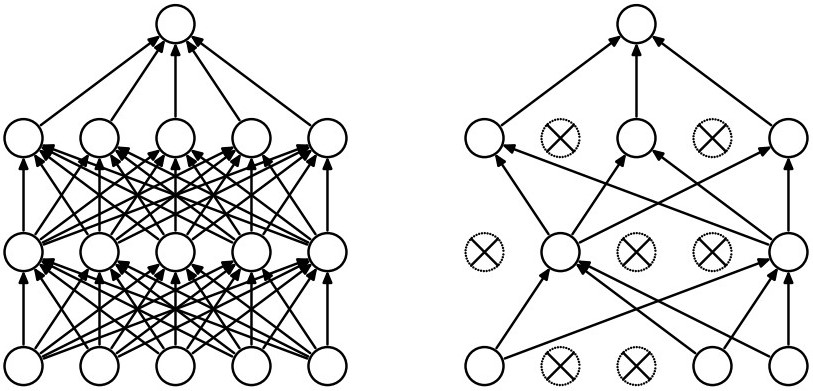
Figure 7: Training loss of 100-layer DenseNet on CIFAR10 using standard learning rate (blue) and M = 6 cosine annealing cycles (red). The intermediate models, denoted by the dotted lines, form an ensemble at the end of training.

- **Snapshot Ensembling**
  Figure 7 depicts the training process using cyclic and traditional learning rate schedules. At the end of each training cycle, it is apparent that the model reaches a local minimum with respect to the training loss. Thus, before raising the learning rate, a "snapshot" of the model weights is taken (indicated as vertical dashed black lines). After training M cycles, we have M model snapshots, $f_1 \dots f_M$, each of which will be used in the final ensemble. It is important to highlight that the total training time of the M snapshots is the same as training a model with a standard schedule (indicated in blue).

- **Ensembling at test time**
  The ensemble prediction at test time is the average of the last m $(m \leq M)$ model's softmax outputs. Let x be a test sample and let $h_i(x)$ be the softmax score of snapshot i. The output of the ensemble is a simple average of the last m models: $h_{Ensemble} = \frac{1}{m} \sum_0^{m-1} h_{M-i}(x)$. We always ensemble the last m models, as these models tend to have the lowest test error.

2.7 Dropout

Neural networks contain multiple non-linear hidden layers and this makes them very expressive models that can learn very complicated relationships between their inputs and outputs. With limited training data, however, many of these complicated relationships will be the result of sampling noise, so they will exist in the training set but not in real test data even if it is drawn from the same distribution.
With unlimited computation, the best way to "regularize" a fixed-sized model is to average the predictions of all possible settings of the parameters, weighting each setting by its posterior probability given the training data. This can sometimes be approximated quite well for simple or small models (Xiong et al., 2011; Salakhutdinov and Mnih, 2008), but we would like to approach the performance of the Bayesian gold standard using considerably less computation. We propose to do this by approximating an equally weighted geometric mean of the predictions of an exponential number of learned models that share parameters.



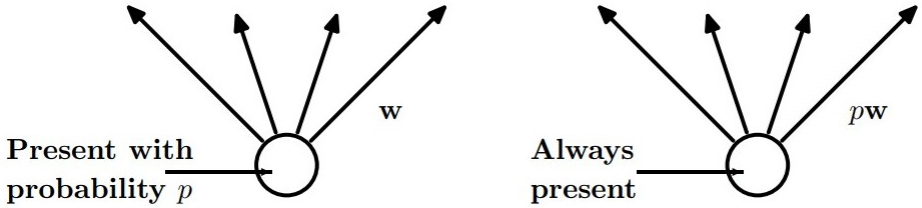
(a) Standard Neural Net (b) After applying dropout.

Figure 8: Dropout Neural Net Model. **Left**: A standard neural net with 2 hidden layers. **Right**: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

Dropout[6] is a technique that addresses both these issues. It prevents overfitting and provides a way of approximately combining exponentially many different neural network architectures efficiently. The term "dropout" refers to dropping out units (hidden and visible) in a neural network. By dropping a unit out, we mean temporarily removing it from the network, along with all its incoming and outgoing connections, as shown in Figure 1. The choice of which units to drop is random. In the simplest case, each unit is retained with a fixed probability p independent of other units, where p can be chosen using a validation set or can simply be set at 0.5, which seems to be close to optimal for a wide range of networks and tasks. For the input units, however, the optimal probability of retention is usually closer to 1 than to 0.5.

Applying dropout to a neural network amounts to sampling a "thinned" network from it. The thinned network consists of all the units that survived dropout . A neural net with n units, can be seen as a collection of 2n possible thinned neural networks. These networks all share weights so that the total number of parameters is still $O(n^2)$, or less. For each presentation of each training case, a new thinned network is sampled and trained. So training a neural network with dropout can be seen as training a collection of 2^n thinned networks with extensive weight sharing, where each thinned network gets trained very rarely, if at all.

At test time, it is not feasible to explicitly average the predictions from exponentially many thinned models. However, a very simple approximate averaging method works well in practice. The idea is to use a single neural net at test time without dropout. The weights of this network are scaled-down versions of the trained weights. If a unit is retained with probability p during training, the outgoing weights of that unit are multiplied by p at test time as shown in Figure 2. This ensures that for any hidden unit the expected output (under the distribution used to drop units at training time) is the same as the actual output at test time. By doing this scaling, 2^n networks with shared weights can be combined into a single neural network to be used at test time.



(a) At training time (b) At test time

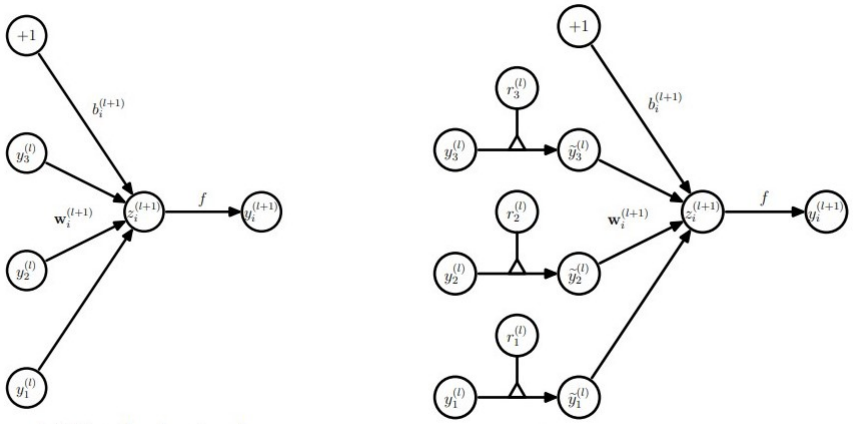
Figure 9: **Left**: A unit at training time that is present with probability p and is connected to units in the next layer with weights w. **Right**: At test time, the unit is always present and the weights are multiplied by p. The output at test time is same as the expected output at training time.

Consider a neural network with L hidden layers. Let $l \in 1, \ldots, L$ index the hidden layers of the network. Let $z^{(l)}$ denote the vector of inputs into layer l, $\mathbf{y}^{(l)}$ denote the vector of outputs from layer l($\mathbf{y}^{(0)} = \mathbf{x}$ is the input). $W^{(l)}$ and $b^{(l)}$ are the weights and biases at layer l. The feed-forward operation of a standard neural network can be described as (for $l \in 0, \ldots, L-1$ and any hidden unit i)

$$z_i^{(l+1)} = \mathbf{w}_i^{(l+1)} \mathbf{y}^l + b_i^{(l+1)}$$
$$y_i^{(l+1)} = f\left(z_i^{(l+1)}\right)$$

where f is any activation function, for example, $f(x) = 1/(1 + exp(-x))$. With dropout, the feed-forward operation becomes

$$r_j^{(l)} \sim \text{ Bernoulli } (p)$$
$$\widetilde{\mathbf{y}}^{(l)} = \mathbf{r}^{(l)} * \mathbf{y}^{(l)}$$
$$z_i^{(l+1)} = \mathbf{w}_i^{(l+1)} \widetilde{\mathbf{y}}^l + b_i^{(l+1)}$$
$$y_i^{(l+1)} = f\left(z_i^{(l+1)}\right)$$



(a) Standard network (b) Dropout network

Figure 10: Caption

Dropout neural networks can be trained using stochastic gradient descent in a manner similar to standard neural nets. The only difference is that for each training case in a mini-batch, we sample a thinned network by dropping out units. Forward and backpropagation for that training case are done only on this thinned network. The gradients for each parameter are averaged over the training cases in each mini-batch. Any training case which does not use a parameter contributes a gradient of zero for that parameter.

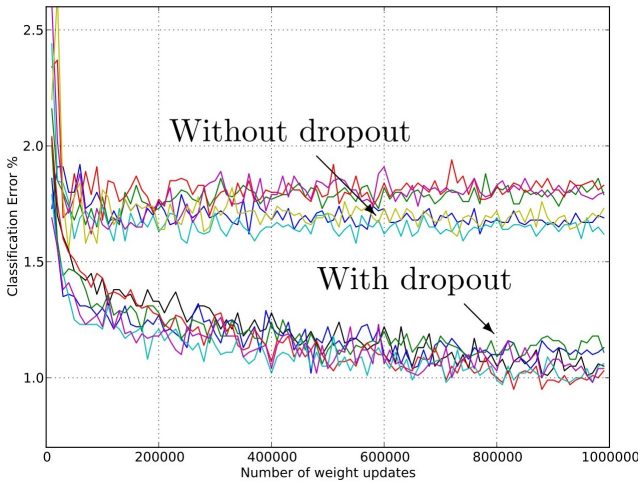


Figure 11: Test error for different architectures with and without dropout. The networks have 2 to 4 hidden layers each with 1024 to 2048 units

2.8 SGD and loss landscape

We Strongly advise to read the following papers in order to develop a theoretical motivation of why smaller batch sizes generalize better and why skip-connections are essential for training extremely deep neural architectures.

1. Finding Flatter Minima with SGD
2. Filter-Wise Normalization for visualizing the loss landscape of Neural Nets

References

[1] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(Jul):2121–2159, 2011.

[2] Gao Huang, Yixuan Li, Geoff Pleiss, Zhuang Liu, John E. Hopcroft, and Kilian Q. Weinberger. Snapshot ensembles: Train 1, get m for free, 2017.

[3] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5mb model size, 2016.

[4] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[5] Tom Schaul, Ioannis Antonoglou, and David Silver. Unit tests for stochastic optimization. *arXiv preprint arXiv:1312.6055*, 2013.

[6] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[7] Ashia C Wilson, Rebecca Roelofs, Mitchell Stern, Nati Srebro, and Benjamin Recht. The marginal value of adaptive gradient methods in machine learning. In *Advances in Neural Information Processing Systems*, pages 4148–4158, 2017.

[8] Hongyi Zhang, Moustapha Cisse, Yann N. Dauphin, and David Lopez-Paz. mixup: Beyond empirical risk minimization, 2017.