University of Illinois Fall 2018

ECE 544NA: Pattern Recognition
Lecture 11: October 2

Lecturer: Alexander Schwing Scribe: Chris Benson

1 Overview

The scribe written below provides a more detailed summary of Lecture 11 for ECE 544 Pattern Recognition taught by Alex Schwing [3]. The lecture covers the topic of **deep learning** using neural networks and **backpropagation** for training deep neural networks.

1.1 Goals

The main goals of this lecture are:

- **Understanding forward and backward pass of neural networks**
- **Learning about backpropagation for training**

1.2 Reading Material

The associated reading material for this lecture is:

- **I. Goodfellow et al.; Deep Learning; Chapters 6-9** [2]

2 Recap

In this class, a general framework for learning has been developed so far. This is displayed in Equation 1.

$$\min_{\mathbf{w}} \frac{C}{2}||\mathbf{w}||_2^2 + \sum_{i\in D}\left(\epsilon \ln \sum_{\hat{y}} \exp \frac{L(y^{(i)},\hat{y}) + \mathbf{w}^\intercal \psi(x^{(i)},\hat{y})}{\epsilon} - \mathbf{w}^\intercal \psi(x^{(i)}, y^{(i)})\right) \tag{1}$$

This is a good learning framework, but it does have issues and limitations. Its main issue is that the feature space of $\psi(x,y)$ is linear. This limits the set of functions that we can learn using this system.

An initial solution proposed to the problem of **linearity** was to use **kernels**. This allows the mapping of the features into other spaces so that you can represent non linear functions. However, this is still learning a model that is **linear** in the parameters $\mathbf{w}$. This is a problem because the model is limited in expressiveness.

Another solution that moves past this problem is replacing the function $\mathbf{w}^\intercal\psi(x,y)$ with a general function $F(\mathbf{w},x,y) \in \mathbb{R}$. This allows the function F to represent and learn non linear functions without the use of kernels. This allows for a more general model learning scheme.

Plugging in this new idea, the learning framework becomes:

$$\min_{\mathbf{w}} \frac{C}{2}||\mathbf{w}||_2^2 + \sum_{i \in D} \left(\epsilon \ln \sum_{\hat{y}} \exp \frac{L(y^{(i)}, \hat{y}) + F(\mathbf{w}, x^{(i)}, \hat{y})}{\epsilon} - F(\mathbf{w}, x^{(i)}, y^{(i)}) \right) \tag{2}$$

This general framework can be used to get to many algorithms including deep learning. This lecture will focus on using this learning framework with deep neural networks and how to train them.

3 Deep Learning

The learning framework defined above now uses a function $F(\mathbf{w}, x, y) \in \mathbb{R}$ such that $y \in (1, 2, 3, ..., k)$.

3.1 Function Choice

A set of good function to choose from for F are the set of differentiable composite functions. More generally, functions that can be represented in an acyclic computation graph and are differentiable are good. This means that all of the functions in the composition are applied in a an acyclic pattern such as linear, and all of them are differentiable.

An example of a possible function is shown in equation 3 and a graphical representation of the function is shown in Figure 3.1.

$$F(\mathbf{w}, x, y) = f_1(w_1, y, f_2(w_2, f_3(...))) \in \mathbb{R} \tag{3}$$

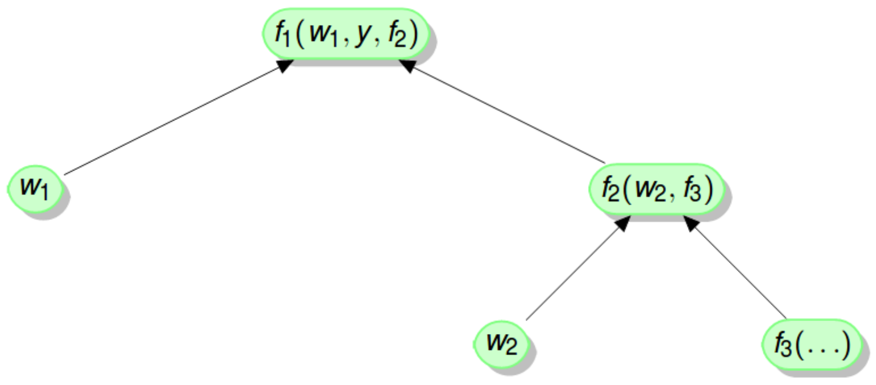


Figure 1: Computational graph of function F [3]

Each of the nodes of the graph are weights, data, and functions. For example, w_1, w_2 are weight node. $f_2(w_2, f_3)$ is the application of function f_2 to weights w_2 and the outputs f_3 from the other nodes in the graph. The final output is the application of f_1 in the top of the acyclic graph. This is how deep learning libraries such as pytorch represent data internally and make computations.

3.2 Layer Choices

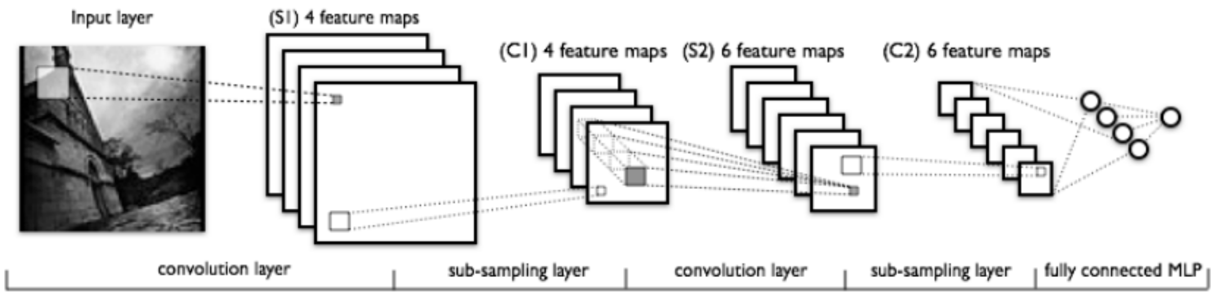
There are a number of different functions that can be used for each of the individual functions in the computational graph. The functions should be differentiable. Below is a list of possible and popular functions:

- **Fully-Connected**
- **Convolutional**
- **Rectified Linear Units (ReLU)**
- **Softmax**
- **Dropout**

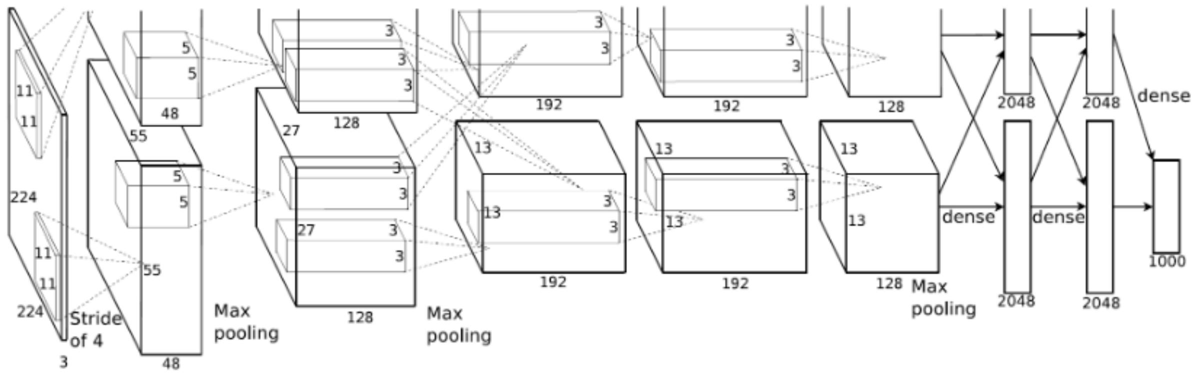
These layers alone can be combined to create complex and effective networks for a broad spectrum of tasks.

3.3 Example Architectures

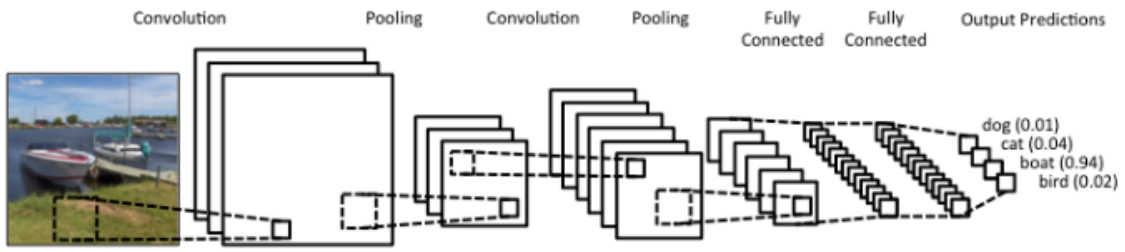
- **LeNet**: The LeNet architecture used convolutional, sub-sampling layers, and fully connected layers. The convolutional layers increased the number of channels and the sub-sampling layers reduced the spatial resolution of the image. This allows for a good network with many fewer parameters.



- **AlexNet**: Alexnet is similar to LeNet in its use of layers. The final output is 1000-dimensional because it is performing a classification task for images with 1000 possible layers. Each index of the 1000 dimensional vector is a score for each of the possible classes.



- **Simple Deep Net**: This neural network applies each layer one after another. Each layers output is used for the next layers input. This does not have to be the case. For example, Residual Neural networks pass combinations of multiple outputs to a single layer as input.



These are example networks that can be trained to achieve very high performance on computer vision tasks. Next, the training process for these networks will be explored.

4 Deep Neural Network Training

For neural networks to be effective, they must be trained to fit some function. Neural networks are commonly are used to predict classes for a set of images. To train a network, a criteria must be used to optimize the network. Starting from the generalized learning framework, the following criteria equation can be formed:

$$\min_{\mathbf{w}} \frac{C}{2}||\mathbf{w}||_2^2 + \sum_{i\in D}\left(\ln \sum_{\hat{y}} \exp F(\mathbf{w}, x^{(i)}, \hat{y}) - F(\mathbf{w}, x^{(i)}, y^{(i)})\right) \tag{4}$$

This equation is formed by setting $\epsilon = 1$ and making the task loss (L) always be equal to 0. This equation is commonly referred to as maximizing the regularized cross entropy and can be written in the following form:

$$\max_{\mathbf{w}} -\frac{C}{2}||\mathbf{w}||_2^2 + \sum_{i\in D}\sum_{\hat{y}} p_{GT}^{(i)}(\hat{y}) \ln p(\hat{y}|x^{(i)}) \tag{5}$$

where GT stands for ground truth and:

$$p_{GT}^{(i)}(\hat{y}) = \delta(\hat{y} = y^{(i)}) \tag{6}$$

$$\delta(\hat{y} = y^{(i)}) = \begin{cases} 1 & \text{if } \hat{y} = y^{(i)} \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

$$p(\hat{y}|x^{(i)}) \propto \exp F(\mathbf{w}, x, \hat{y}) \tag{8}$$

The C is in this equation is a regularization constant that facilitates weight decay. This can be shown below:

$$\min_{\mathbf{w}} \underbrace{\frac{C}{2}||\mathbf{w}||_2^2}_{\text{weight decay}} \underbrace{- \sum_{i\in D}\sum_{\hat{y}} p_{GT}^{(i)}(\hat{y}) \ln p(\hat{y}|x^{(i)})}_{\ell(\text{gt,F})} \tag{9}$$

The first part of the question is the weight decay and the second part is the loss between the ground truth and the predicted F values.

4.1 Optimizing with Gradient Descent

With a defined criteria equation, the network must be trained to solve this optimization problem. This can be done using **stochastic gradient descent (SGD) with momentum**. SGD with momentum is an optimization technique involving repeatedly moving down along the gradient of a batch of examples. This will move the function in a direction that will minimize the function. Adding momentum causes SGD to take into account past gradients to end up moving faster in directions that are repeatedly traveled and slow in directions that are contradicted by adjacent gradients. This can be compared to rolling a heavy ball down an n-dimensional hill. The update rule for SGD with momentum can be formally defined as:

$$\mathbf{V}_{k+1} = \mathbf{v}_k + \nabla f(\mathbf{w}_k) \tag{10}$$

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \alpha \mathbf{v}_{k+1} \tag{11}$$

where f is the a loss function and α is a learning rate parameter. The loss function being used here for f is equation 4. $\mathbf{v}$ takes is a sum of past gradients. This causes the update to $\mathbf{w}$ to take into account past gradient in its update. To preform the SGD update of $\mathbf{v}$ and $\mathbf{w}$, the gradient must be computed for equation 4. This is shown below:

$$\nabla f(\mathbf{w}) = C\mathbf{w} + \sum_{i \in D} \sum_{\hat{y}} \left(p(\hat{y}|x^{(i)}) - \delta(\hat{y} = y^{(i)}) \right) \frac{\partial F(\mathbf{w}, x^{(i)}, \hat{y})}{\partial \mathbf{w}} \tag{12}$$

Now, this value must be computed to perform gradient descent. $C\mathbf{w}$ is easy to compute because both are known numeric values. $\delta(\hat{y} = y^{(i)})$ is also simple to compute based of equation 7 which basically checks if the predicted label equals ground truth label. $p(\hat{y}|x^{(i)})$ can be computed using Softmax of the logits produced by F. The calculation is shown below:

$$p(\hat{y}|x^{(i)}) = \frac{\exp F(\mathbf{w}, x, \hat{y})}{\sum_{\tilde{y}} \exp F(\mathbf{w}, x, \tilde{y})} \tag{13}$$

The final part of this equation that must be calculated is $\frac{\partial F(\mathbf{w}, x^{(i)}, \hat{y})}{\partial \mathbf{w}}$ which will be calculated using the backpropagation algorithm.

4.2 Backpropagation

Backpropagation is an efficient way to compute gradients of a deep neural networks by passing intermediate gradient values down the acyclic computational graph to improve computational time. To demonstrate this algorithm, an example computation will be one for the equation:

$$F(\mathbf{w}, x, y) = f_1(w_1, y, f_2(w_2, f_3(w_3(x)))) \text{ with activations} \begin{cases} x_2 = f_3(w_3, x) \\ x_1 = f_2(w_2, x_2) \end{cases} \tag{14}$$

Now, to compute $\frac{\partial F(\mathbf{w}, x^{(i)}, \hat{y})}{\partial \mathbf{w}}$, each of the partial derivative with respect to $\mathbf{w}$ must computed. This can be done using the chain rule. For example:

$$\frac{\partial F(\mathbf{w}, x^{(i)}, \hat{y})}{\partial \mathbf{w_3}} = \frac{\partial f_1}{\partial x_1} \frac{\partial x_1}{\partial x_2} \frac{\partial x_2}{\partial w_3} = \underbrace{\frac{\partial f_1}{\partial f_2}} \frac{\partial f_2}{\partial f_3} \frac{\partial f_3}{\partial w_3} \tag{15}$$

$$\frac{\partial F(\mathbf{w}, x^{(i)}, \hat{y})}{\partial \mathbf{w_2}} = \frac{\partial f_1}{\partial x_1} \frac{\partial x_1}{\partial w_2} = \underbrace{\frac{\partial f_1}{\partial f_2}} \frac{\partial f_2}{\partial w_2} \tag{16}$$

One take away from this is that in computing the partial derivative using chain rule, some partial derivatives are calculated multiple times. Above, $\frac{\partial f_1}{\partial f_2}$ is computed for both the partial derivative with respect to w_2 and w_3.

Doing backpropagation on functions that can be represented as acyclic computational graphs, allow for efficient gradient computation. Partial derivative can be passed down the computational graph so that derivatives do not need to calculated multiple times. A visual display of gradient passing in backpropagation is shown in Figure 2 for equation 14.

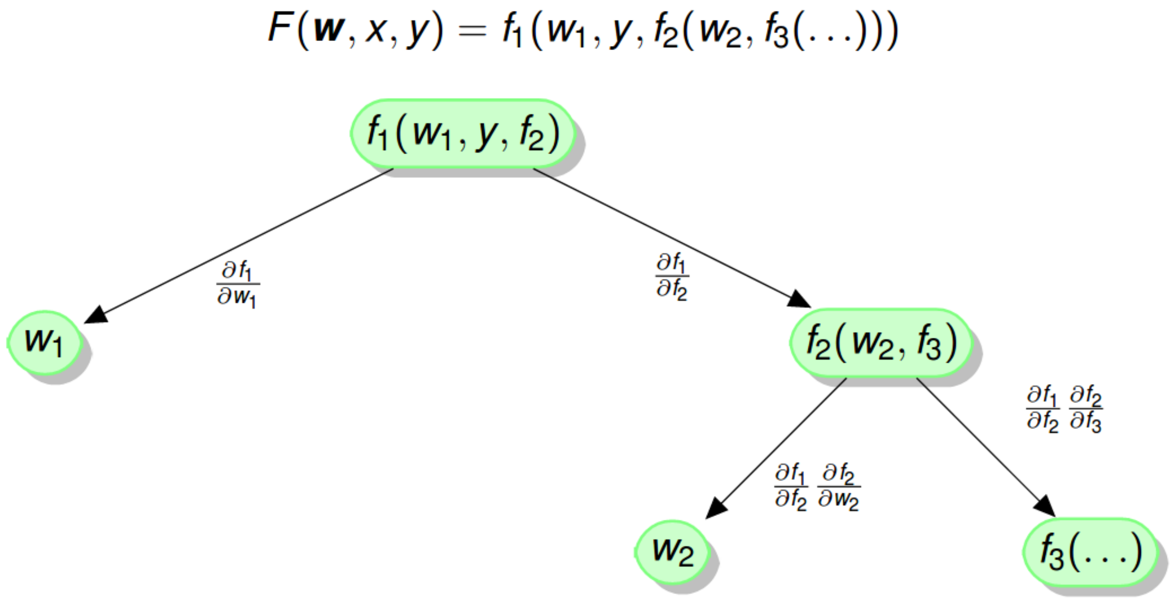

$F(\mathbf{w}, x, y) = f_1(w_1, y, f_2(w_2, f_3(\ldots)))$
$f_1(w_1, y, f_2)$
$\frac{\partial f_1}{\partial w_1}$
$\frac{\partial f_1}{\partial f_2}$
w_1
$f_2(w_2, f_3)$
$\frac{\partial f_1}{\partial f_2} \frac{\partial f_2}{\partial f_3}$
$\frac{\partial f_1}{\partial f_2} \frac{\partial f_2}{\partial w_2}$
w_2
$f_3(\ldots)$

Figure 2: Backpropagation of gradients in the computational graph for F [3]

Since $\frac{\partial f_1}{\partial f_2}$ is used by both nodes f_1 and f_2, it can be first computed by f_1 and passed down to f_2 to prevent repeat calculations. This can be repeated all the way down the graph for all nodes.

This allows efficient computation for $\frac{\partial F(\mathbf{w}, x^{(i)}, \hat{y})}{\partial \mathbf{w}}$. Once this value is obtained, the value for $\nabla f(\mathbf{w})$ is completely calculated. Now, the value for $\mathbf{w}$ can be updated based on equations 10 and 11. After repeating this process for a number of batches of data, the function should start to approach a minimum allowing for the deep neural network to behave as a useful predictor.

4.3 Information Storage

During the forward pass, sometimes values need to be stored at each node in the graph.

- **Inference**: Inference is when values for F are being calculated, but training is not occurring. In this case, nothing needs to be stored because backpropagation will not occur. Not storing intermediate results makes this process faster. This means that once a network is trained inference can be performed quickly.
- **Learning**: Learning is when values for F are being calculated and backpropagation is occurring after each batch of training data. In this case, intermediate values for fully connected, convolutional, and other non in-place functions should be stored. These values are required to compute the gradient. However, some functions such as ReLU and other activation functions

do not need to store the intermediate values and can be combined with the previous layer. This reduces the amount of information needed to be stored. This is the difference between activation functions and layers.

4.4 Remark on Constraints

In the formulation of $F(\mathbf{w}, x, y)$ for deep neural networks, the function is not constrained in any form. This means that the loss function is likely no longer convex. This has two main implications:

- Gradient descent is no longer guaranteed to converge to a global optimal. This is important because it is no longer garunteed to find the best optimal solution. The optimization could get stuck in a local minimum.
- The initialization of $\mathbf{w}$ is matters for finding the best minimum. With a good initialization, the optimization is less likely to get stuck at a local minimum.

4.5 Initialization

The initialization of $\mathbf{w}$ matters for deep learning performance. However, proper initialization is not well understood. It is known that the initialization must break symmetry. A couple common initialization schemes are:

- Random Uniform
$$\text{Uniform}\left(-\frac{1}{\sqrt{\text{fan in}}}, \frac{1}{\sqrt{\text{fan in}}}\right)$$
- Glot and Bengio (2010) (Xavier Initialization)
$$\text{Uniform}\left(-\frac{6}{\sqrt{\text{fan in} + \text{fan out}}}, \frac{6}{\sqrt{\text{fan in} + \text{fan out}}}\right)$$

Fan in is defined as the number of input features to a layer and fan out is defined as the number of output features of a layer.

5 Traits of Neural Networks

Neural networks are can solve a large number of complex optimization problems. Neural networks have some positive and negative traits that are interesting to consider.

- A deep neural network with a single connected layer is equivalent to logistic regression. This can be derived from the general learning framework.
- Deep neural networks do not use hand-crafted features. Up until around 2012, most machine learning models used hand-crafted features to achieve good performance on tasks. Neural networks revolutionized this by automatically learning feature space transformations that allowed for good performance on tasks. Neural networks can learn hierarchical abstractions of data that were previously quite challenging to compute. Figure 3 shows how neural networks extract features. The early layers tend to extract small features such as eyes, noses, or ears, and then later layers tend to extract larger, hierarchical features such as a whole face.
- Neural networks are computationally demanding and typically require GPUs. The large number of features in neural networks results in large amounts of compute being needed for neural networks to be trained to achieve good performance. This can be accelerated using GPUs, but it is requires a large amount of resources.

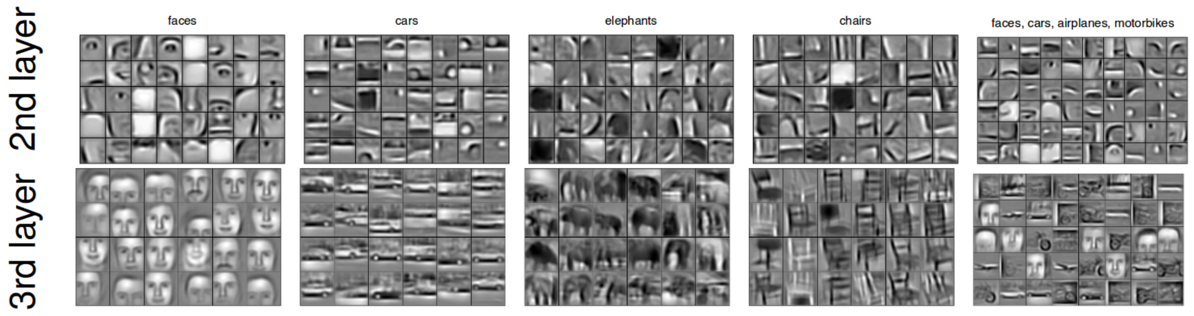


Figure 3: Example features extracted from different layers of a deep neural network

6 Recent Popularity

Currently, deep neural networks are incredibly popular and are being used to try to solve almost every problem in machine learning. This popularity is due to a number of reasons.

First, neural networks require tremendous amounts of computational resources and training data to perform better than simpler models. The ideas of neural networks have been around for a long time, but only now are large enough data sets and compute power available for them to be effective. Figure 4 demonstrates how neural networks with more data and compute are outperforming older approaches.

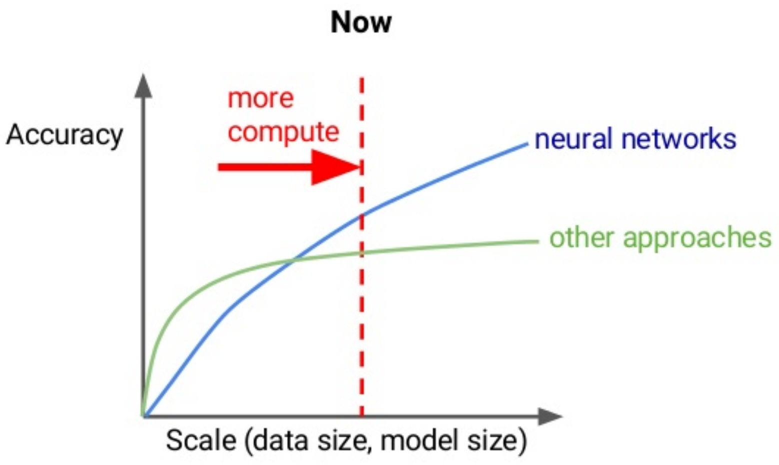


Figure 4: Additional compute and data has made deep learing outperform other methods [1].

6.1 Algorithmic Advances

Secondly, there have been a number of algorithmic advances that have enabled improved performance in recent years. These include:

- **Rectified Linear Unit (ReLU)**: ReLU = max(x,0). ReLU is a more recent activation function that fixes the vanishing gradient problem for lower layers of neural networks. The vanishing gradient problem occurs when gradients become 0 at lower layers of a neural networks causing lower layers to not be trained effectively. This would occur with sigmoid activation function because it saturates to a maximum value of 1. When the max value goes to 1, the gradient is saturated and cannot get higher. As this backpropagates, the gradient becomes 0. ReLU rectifies this because in the positive direction ReLU is linear and can get values much higher than 1.

- **Dropout**: Dropout is a training technique where some neurons are randomly set to 0 on the forward pass. This causes more neurons to be needed in training and a complete learning of features. This decorrelates units and causes them to learn different features.
- **Good Initialization**: Better initialization heuristics have helped neural networks from getting stuck in local minimums instead of approaching global optimums.
- **Batch Normalization**: During training, data at each layer is normalized by subtracting mean and dividing by standard deviation. This allows each layer to learn individually instead of being dominated by large or small values being passed from earlier layers.

7 Choices for Implementing Deep Neural Networks

There are a few important areas to consider when implementing a Deep neural networks.

7.1 Network Design

It is important to design a good composite function $F(\mathbf{w}, x, y)$ to be your network. Make sure, it is understood what all layers do and understand the dimensions of the inputs and outputs of each layers. Consider using convolutional networks for image or video data. Think about using recurrent neural networks for natural language problems.

7.2 Choice of Loss Function

There are many different loss functions that can be used to train neural networks. It is important to understand when to use different loss functions and what inputs and targets they expect.

- **Cross Entropy Loss**: Typically used for classification tasks with many classes. x should be logits (non-normalized outputs)

$$loss(x, class) = -\log\left(\frac{\exp(x[class])}{\sum_j \exp(x[j])}\right) = -x[class] + \log(\sum_j \exp(x[j]))$$

- **Negative Log Likelihood Loss (NLLLoss)**: Use for classification tasks with many classes. x should be normalized probabilities

$$loss(x, class) = -x[class]$$

- **Mean Squared Error Loss**: Use for regression problems. y is a number not a class in this case.

$$loss(x, y) = \frac{1}{n}\sum_i |x_i - y_i|^2$$

- **Binary Cross Entropy Loss (BCE)** : Use for binary classification tasks. Inputs should be normalized probabilities The summation will always simplify to one term because either t[i] or 1-t[i] will be 0.

$$loss(o,t) = -\frac{1}{n}\sum_i i(t[i] * \log(o[i]) + (1 - t[i]) * \log(1 - o[i]))$$

- **Binary Cross Entropy with Logits Loss** : Same as BCE Loss but adds a sigmoid layer to normalize. Inputs are logits.

$$loss(o,t) = -\frac{1}{n}\sum_i i(t[i] * \log(sigmoid(o[i])) + (1 - t[i]) * \log(1 - sigmoid(o[i])))$$

- **L1 Loss**: simple L1 norm.
- **Kullback-Leibler Divergence Loss**: Used for measuring differences in continuous distributions.

It is important to pass the expected inputs into the appropriate loss functions. For example, NLLLoss expects inputs to be in the form of log likelihoods. The input to NLLLoss should come from a log Softmax layer:

$$f_i(x) = \log \frac{\exp x_i}{\sum_j \exp x_j} \tag{17}$$

This is important because log Softmax layers can compute these values while maintaining numerical stability. The log sum exp trick allows the log of a sum of exponentials to be converted to:

$$\log \sum_j \exp x_j = c + \log \sum_j \exp(x_j - c) \tag{18}$$

If c is set to the maximum of all x_j values, this should accurately calculate the maximum value of log sum exp with more stability. Without this trick, computation will be unstable and NLLLoss will not train properly.

8 Popular Deep Learning Architectures and Datasets

Below is a list of important deep learning architectures and datasets:

8.1 Architectures

- LeNet: (above)
- AlexNet: (above)
- VGG (16/19, 3x3 convolutions): One of the earlier very deep models
- GoogLeNet: Introduced the inception module which used 1x1 convolutions.
- ResNet: Introduced residual connections which allows skipping of intermediate layers of a network and helps prevent vanishing gradient problem for really deep networks.

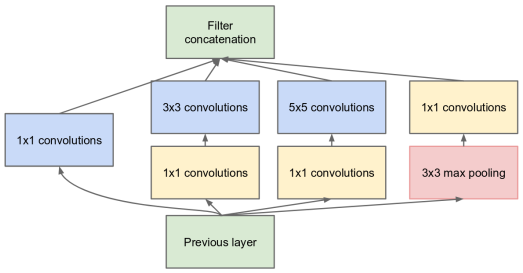


Figure 5: Inception module of Googlenet

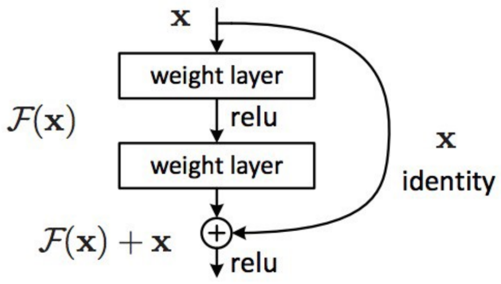


Figure 6: Residual Layer of Resnet

8.2 Imagenet Challenge

The Imagenet Challenge is a large dataset of 1.2M images in 1000 classes. The task was supposed to be very difficult with the goal of challenging human vision abilities. This was a bench mark goal for computer vision for many years.

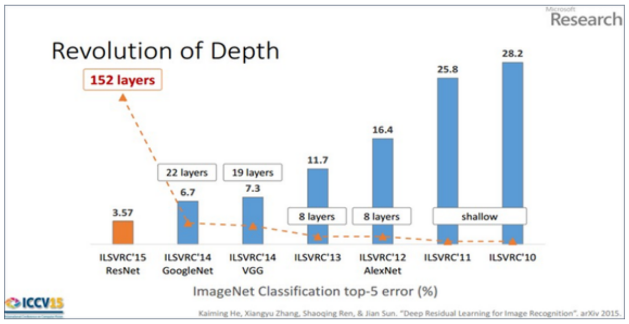


Figure 7: Graph showing depth leading to greater performance on ImageNet Challenge Dataset [3]

A few years ago, neural networks able to finally achieve better than human performance on this task. Alexnet revolutionized performance by using deep neural networks on a GPU to perform better than previous state of the art. Another key breakthrough was the use of ReLU units to prevent vanishing gradient and to simply optimization. Very deep networks were the final breakthrough that pushed performance past human levels as shown in Figure 7.

9 Quiz

- **What are deep nets?**
  A pattern recognition technique that uses compositions of differentiable functions, typically in an acyclic computational graph, to perform classification or regression tasks.
- **How do deep nets relate do SVMs and logistic regression?**
  - SVM are classifiers that can separate linear data or using kernels non-linear data. Deep nets can separate non-linear data without kernel methods.
  - Logistic regression is a simplified case of deep learning. A neural network with 1 fully connected layer is performing logistic regression.

- **What is back-propagation in deep nets?**
  A method for propagating gradients of the network efficiently through the network.
- **What components of deep nets do you know?**
  Fully connected layer, ReLU activation layer, Cross Entropy Loss, etc.
- **What algorithms are used to train deep nets?**
  SGD with momentum using backprop. etc.

References

[1] J. Dean. Trends and Developments in Deep Learning Research, January 2017.

[2] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning.* The MIT Press, 2016.

[3] A. Schwing. ECE 544: Pattern Recognition Lecture 11: Backprop (deep nets), October 2018.